Dynamic Formal Verification of Pthread / C programs

About 45 minutes – by Ganesh

Dynamic Verification is promising

- Static Analysis
  - Scalable but false positive rate can be high
- Dynamic Detection
  - Scalable but may fail to expose all concurrency bugs
- Formal Static Verification
  - Still has a long way to apply to real applications due to the limitations modeling and decision procedures.
- Dynamic Verification
  - Concretely execute the program
  - Guarantee concurrency bug exposure
  - No false alarms

Growing Importance of Dynamic Verification

Code written using mature libraries (MPI, OpenMP, PThreads, …)

API calls made from real programming languages (C, Fortran, C++)

Runtime semantics determined by realistic compilers and runtimes

Dynamic Verification Methods are going to be very important for real engineers !

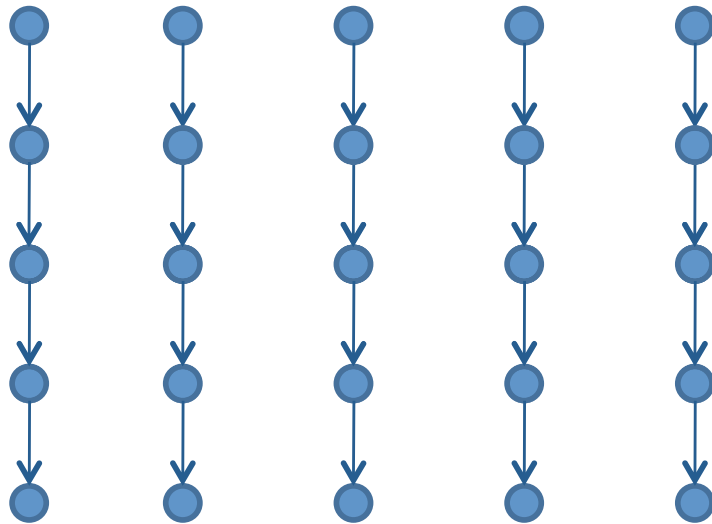
(static analysis and model based verification can play important supportive roles)

Need Efficient Dynamic Verification Methods

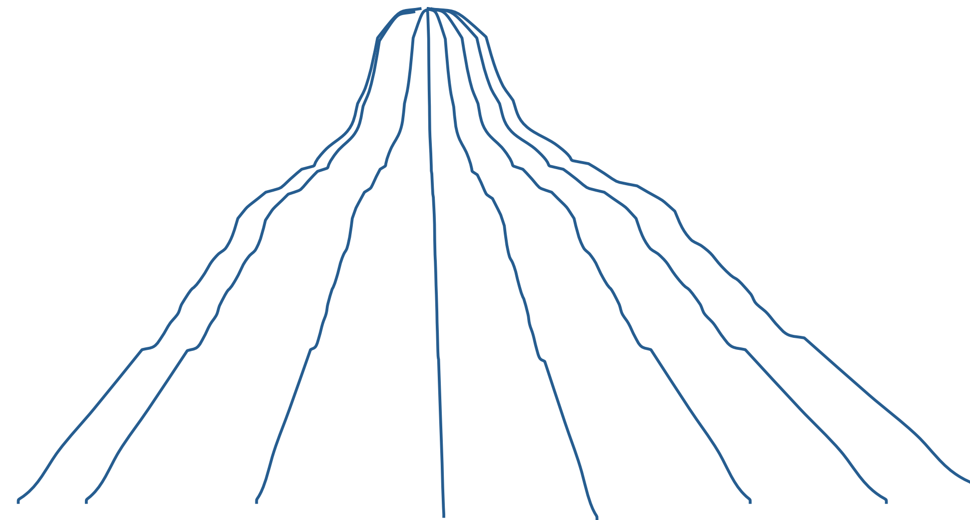
- Dynamic verification faces the state explosion problem
  - Number of possible executions grows exponentially while the program's size increases
  - 5 steps, 5 threads → 10 billion interleavings

    $(nk)!/(k!)^n \qquad (5*5)!/(5!)^5 > 10\, billion$
  - This causes severe omissions in practice
- Efficient dynamic verification algorithms are essential to make this technique applicable to general concurrent programs.

Exponential number of TOTAL Interleavings – most are EQUIVALENT – generate only RELEVANT ones !!

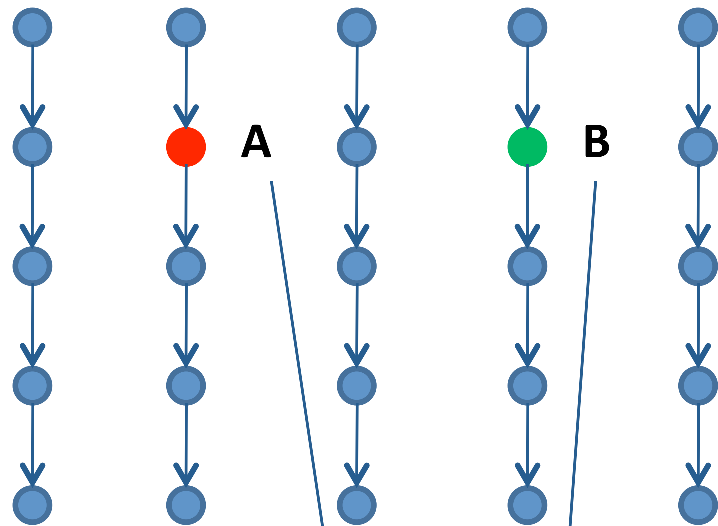


TOTAL > 10 Billion Interleavings !!



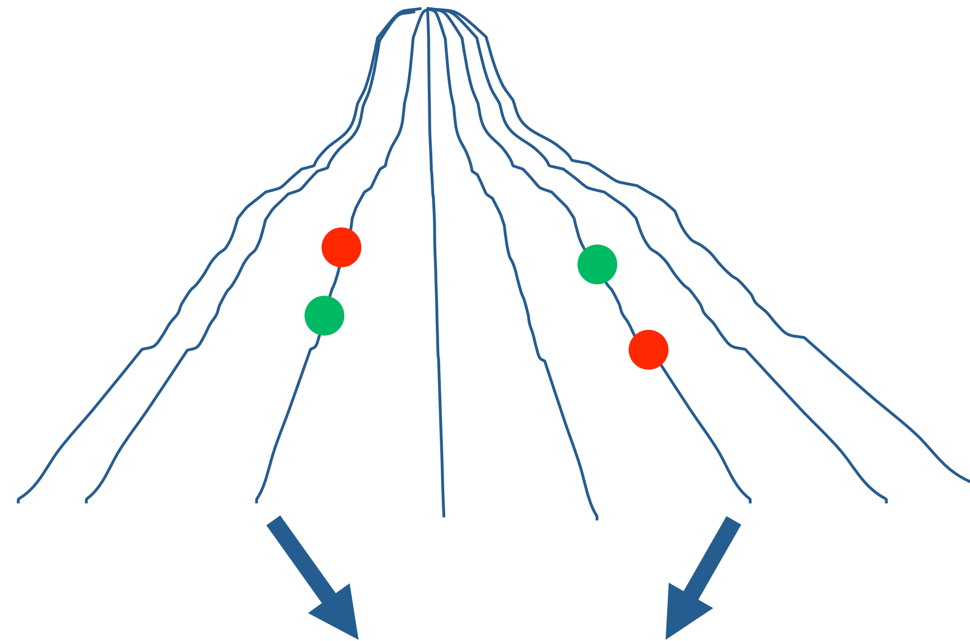
Inspect's Dynamic Partial Order Reduction

TOTAL > 10 Billion Interleavings !!



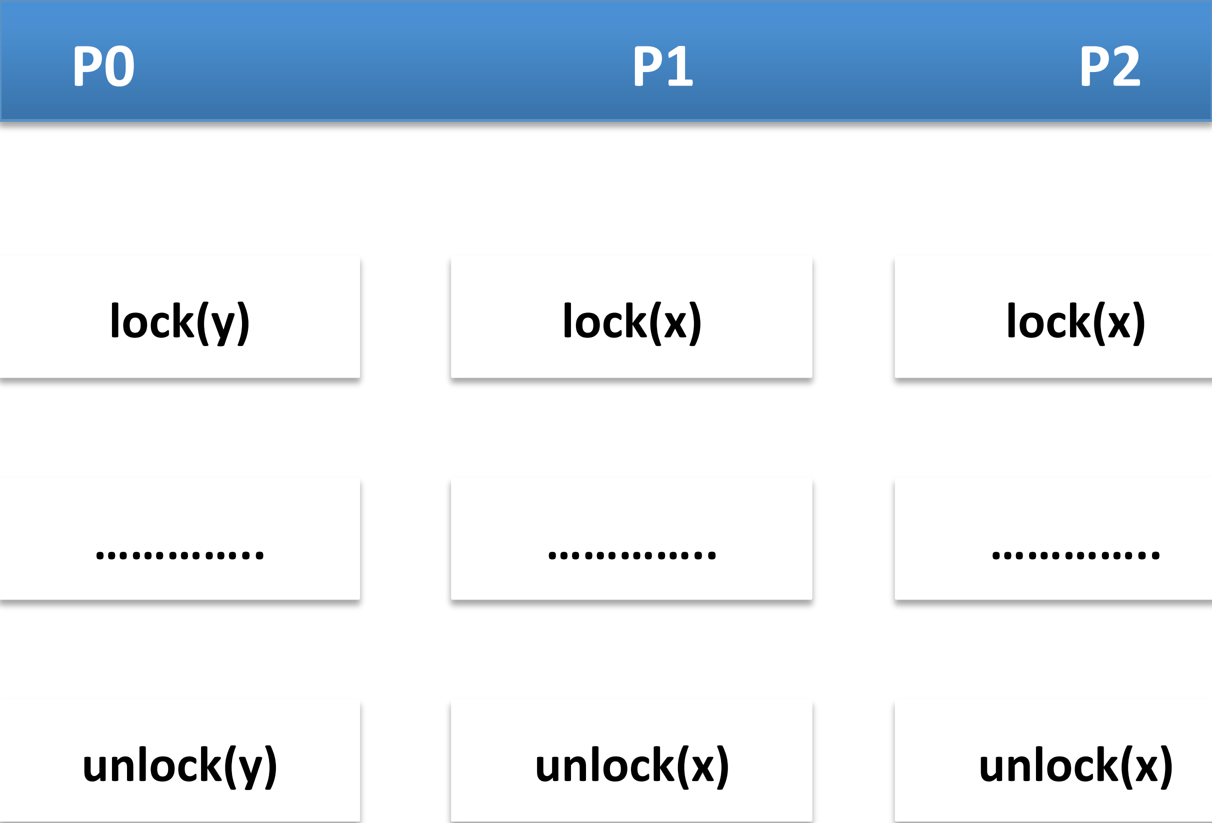
Dependent actions (e.g., x++ and x--, or accesses to the same lock)

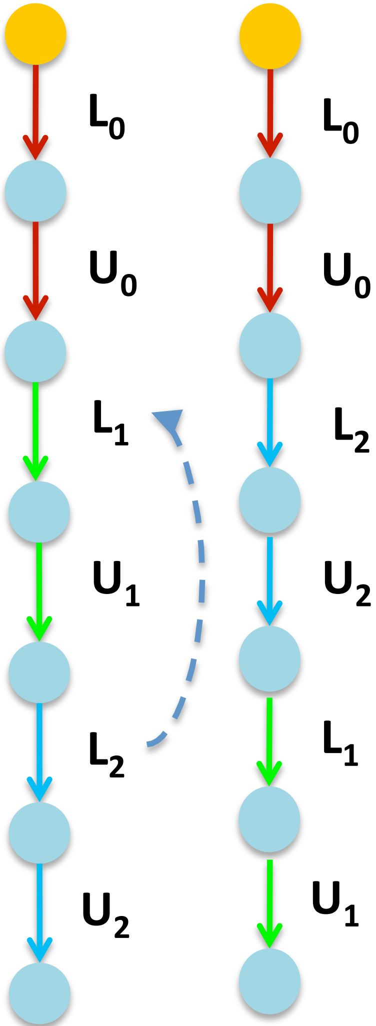
All other actions are pairwise independent



Only these 2 are RELEVANT!!!

Overview of stateless dynamic verification





Contributions of the Inspect project

- A family of algorithms for efficient dynamic verification
  - Stateful dynamic partial order reduction
  - Property-driven pruning for fast race detection
  - Symmetry discovery using dynamic analysis
  - Distributed dynamic partial order reduction
- Inspect – a dynamic verification framework
  - Combine program analysis, program instrumentation and model checking in a unique way to realize efficient dynamic verification

Features of Inspect

- Inspect is a tool for finding (with guarantee, for a test harness)
  - Deadlocks
  - Data races
  - Assertion violations

  in C/Pthread programs
- Only DPOR-based tool of its kind
- Available for free download
- Has an elaborate static analysis front-end that we have developed using Berkeley CIL

Example with Deadlock

```
void * thread_A(void* arg)
{
 pthread_mutex_lock(&mutex);
 A_count++;
 if (A_count == 1)
  pthread_mutex_lock(&lock);
 pthread_mutex_unlock(&mutex);

 pthread_mutex_lock(&mutex);
 A_count--;
 if (A_count == 0)
  pthread_mutex_unlock(&lock);
 pthread_mutex_unlock(&mutex);
}
```

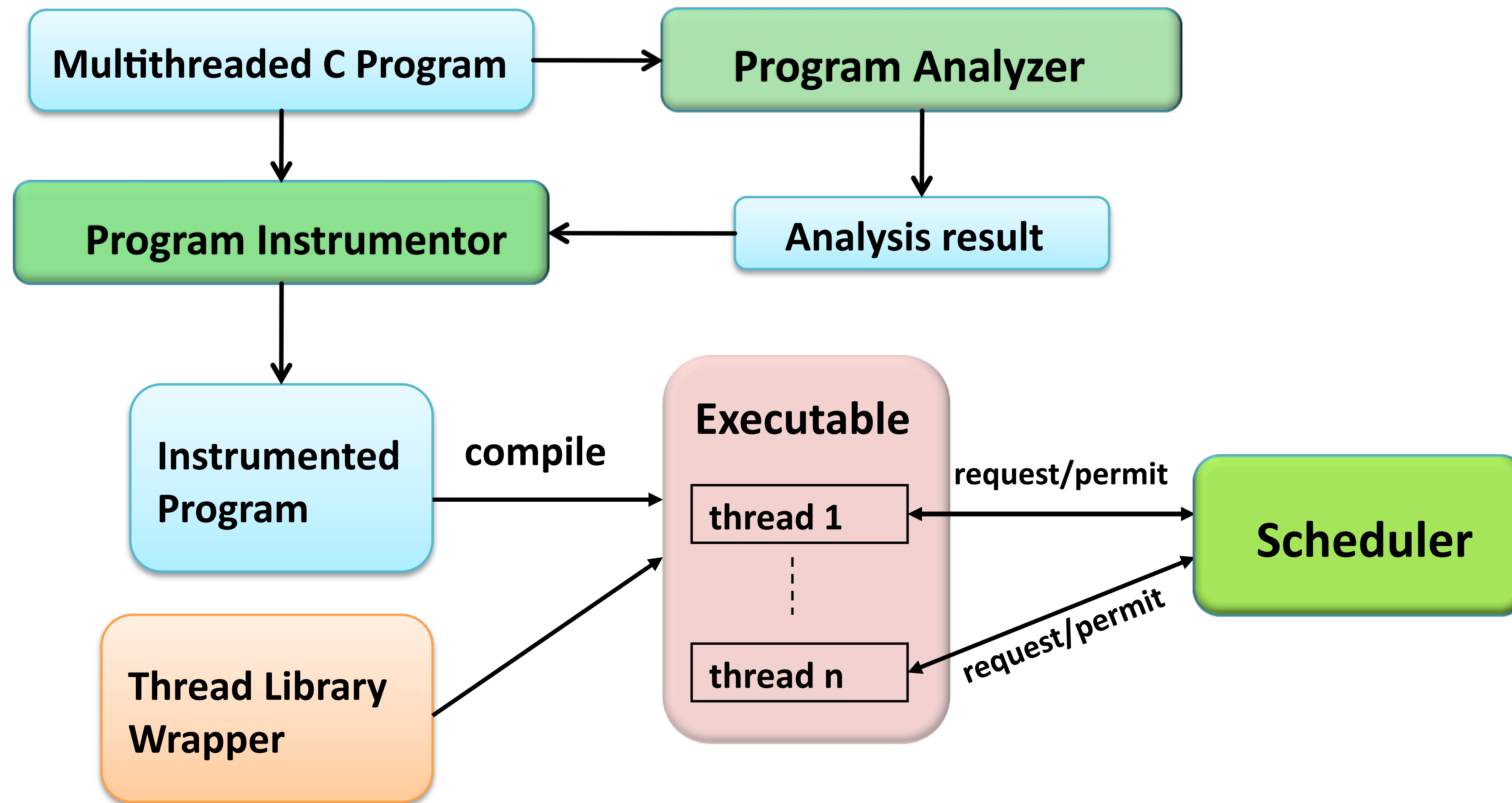
```
void * thread_B(void * arg)
{
 pthread_mutex_lock(&mutex);
 B_count++;
 if (B_count == 1)
  pthread_mutex_lock(&lock);
 pthread_mutex_unlock(&mutex);

 pthread_mutex_lock(&mutex);
 B_count--;
 if (B_count == 0)
  pthread_mutex_unlock(&lock);
 pthread_mutex_unlock(&mutex);

}
```

Summary of Inspect Commands and UI

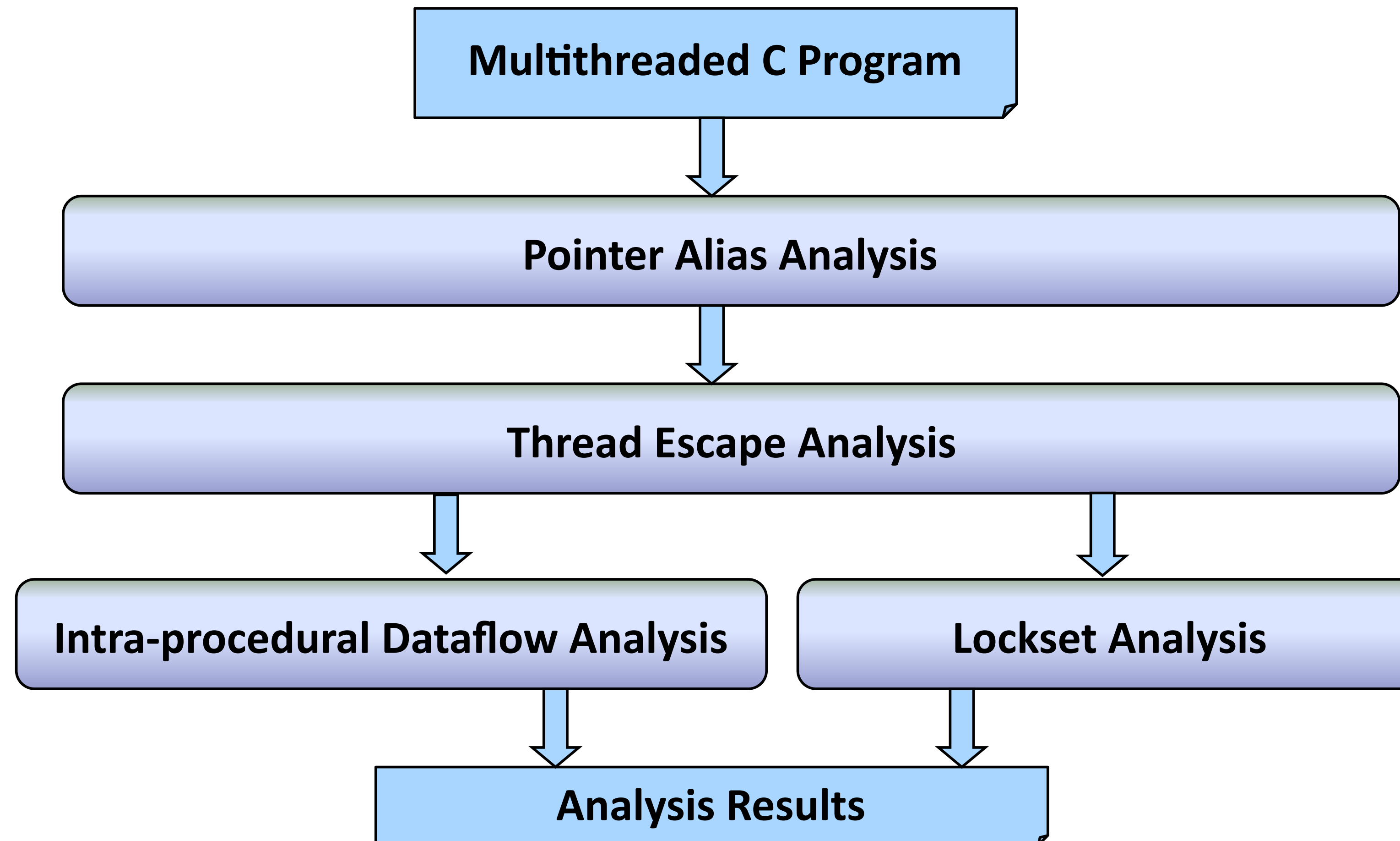
- Go to the working directory of Inspect
- Type these:
  - bin/instrument simple-pthread-deadlock.c
  - Bin/compile simple-pthread-deadlock.instr.c
  - ./inspect ./target
  - Follow user manual to debug

- Use prelim Emacs user interface to launch Inspect and also view various traces

Application of Dynamic Verification: The Inspect Project

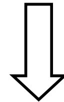


Program Analysis within Inspect



Source code transformation (1)

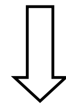
functions calls to the thread library routine



functions calls to the Inspect library wrapper

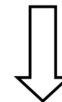
In detail:

pthread_create



inspect_thread_create

pthread_mutex_lock



inspect_mutex_lock

…

Source code transformation (2)

```
x =  rhs;
```

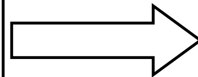
```
write_shared_xxx(&x, rhs);
.....
void write_shared_xxx(type * addr, type val){
    inspect_obj_write(addr);
    *addr = val;
}
```

```
lhs =  x;
```

```
read_shared_xxx(&lhs, &x);
.....
void read_shared_xxx(type * lhs, type * addr){
    inspect_obj_read(addr);
    *lhs = *addr;
}
```

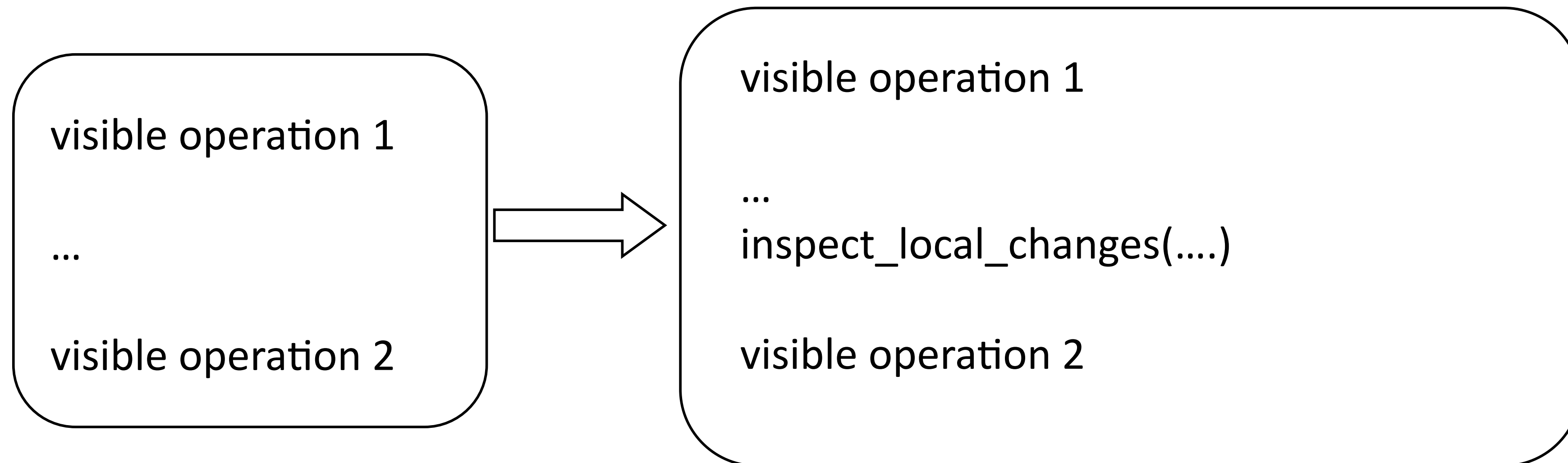
Source Transformation (3)

```
thread_routine(…){

…

}
```



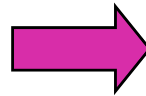
```
thread_routine(…){
  inspect_thread_begin();

  …

  inspect_thread_end();
}
```

Source Transformation (4)



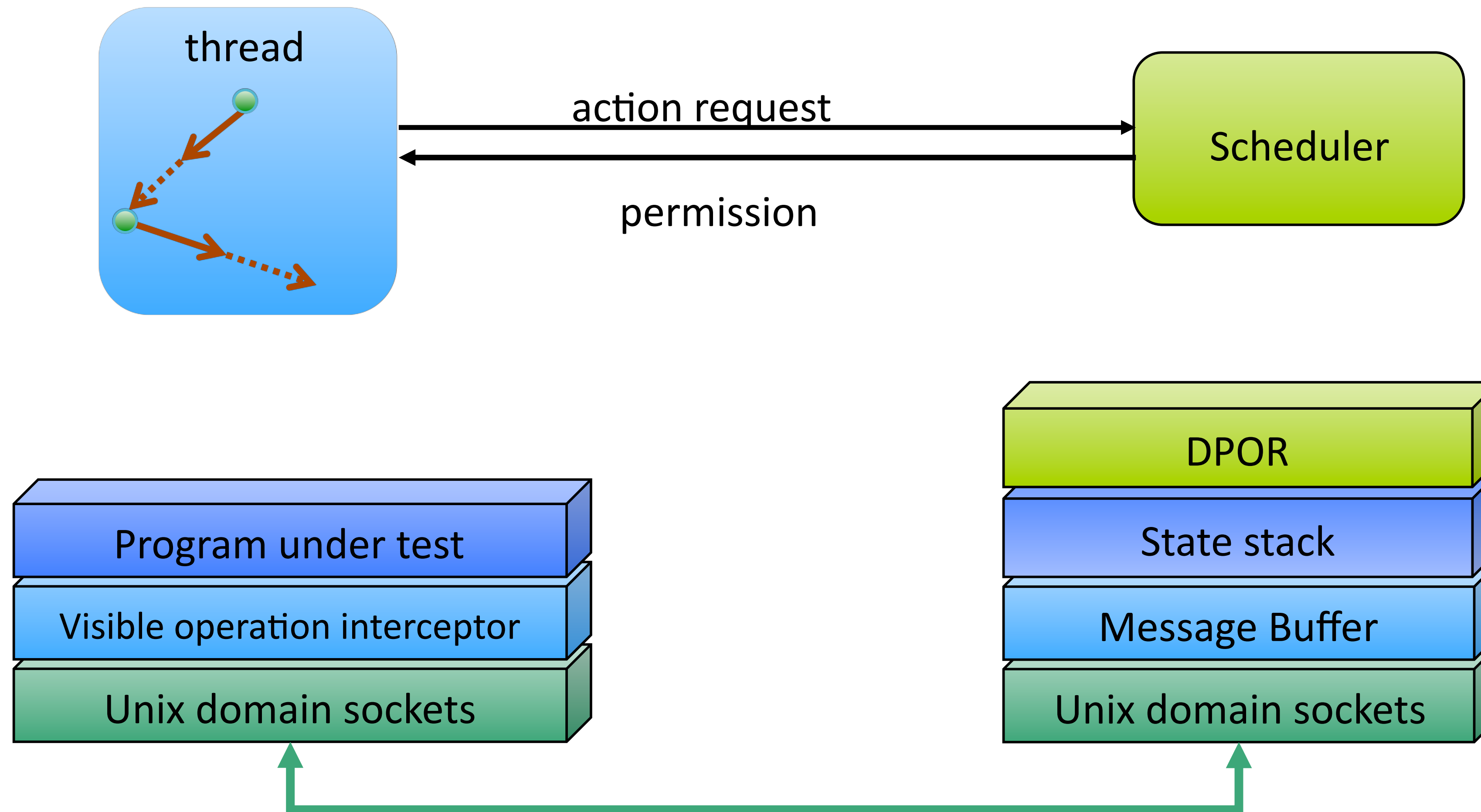
Result of instrumentation

```
void * Philosopher(void * arg){
  int i;
  i = (int)arg;
  ...
  pthread_mutex_lock(&mutexes[i%3]);
  ...
  while (permits[i%3] == 0) {
    printf("P%d : tryget F%d\n", i, i%3);
    pthread_cond_wait(...);
  }
  ...
  permits[i%3] = 0;
  ...
  pthread_cond_signal(&conditionVars[i%3]);

  pthread_mutex_unlock(&mutexes[i%3]);
  return NULL;
}
```



```
void *Philosopher(void *arg )
{ int i ;
  pthread_mutex_t *tmp ;
  {
  inspect_thread_start("Philosopher");
  i = (int )arg;
  tmp = & mutexes[i % 3]; ...
  inspect_mutex_lock(tmp); ...
  while (1) {
    __cil_tmp43 = read_shared_0(& permits[i % 3]);
    if (! __cil_tmp32) {
      break;
    }
    __cil_tmp33 = i % 3; ...
    tmp___0 = __cil_tmp33; ...
    inspect_cond_wait(...);
  }
  ...
  write_shared_1(& permits[i % 3], 0);
  ...
  inspect_cond_signal(tmp___25);
  ...
  inspect_mutex_unlock(tmp___26);
  ...
  inspect_thread_end();
  return (__retres31);
}
```

Inspect animation



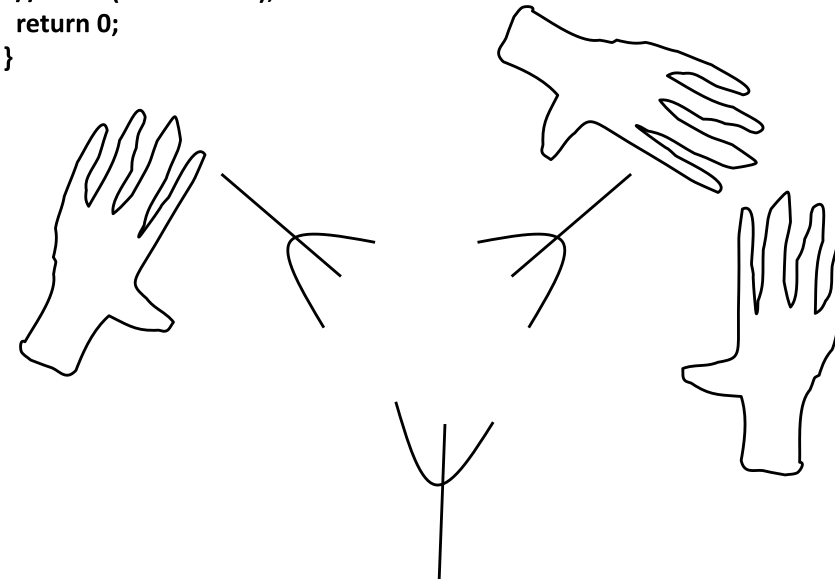
Example of Dining Philosophers

```
#include <stdlib.h>    // Dining Philosophers with no deadlock
#include <pthread.h>   // all phils but "odd" one pickup their
#include <stdio.h>     // left fork first; odd phil picks
#include <string.h>    // up right fork first
#include <malloc.h>
#include <errno.h>
#include <sys/types.h>
#include <assert.h>

#define NUM_THREADS 3

pthread_mutex_t mutexes[NUM_THREADS];
pthread_cond_t conditionVars[NUM_THREADS];
int permits[NUM_THREADS];
pthread_t tids[NUM_THREADS];

int data = 0;

void * Philosopher(void * arg){
  int i;
  i = (int)arg;

  // pickup left fork
  pthread_mutex_lock(&mutexes[i%NUM_THREADS]);
  while (permits[i%NUM_THREADS] == 0) {
    printf("P%d : tryget F%d\n", i, i%NUM_THREADS);
    pthread_cond_wait(&conditionVars[i%NUM_THREADS],&mutexes[i
%NUM_THREADS]);
  }
```

```
permits[i%NUM_THREADS] = 0;
  printf("P%d : get F%d\n", i, i%NUM_THREADS);
  pthread_mutex_unlock(&mutexes[i%NUM_THREADS]);

  // pickup right fork
  pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
  while (permits[(i+1)%NUM_THREADS] == 0) {
    printf("P%d : tryget F%d\n", i, (i+1)%NUM_THREADS);
    pthread_cond_wait(&conditionVars[(i+1)%NUM_THREADS],&mutexes[(i
+1)%NUM_THREADS]);
  }
  permits[(i+1)%NUM_THREADS] = 0;
  printf("P%d : get F%d\n", i, (i+1)%NUM_THREADS);
  pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);

  //printf("philosopher %d thinks \n",i);
  printf("%d\n", i);

  //  data = 10 * data + i;

  fflush(stdout);


  // putdown right fork
  pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
  permits[(i+1)%NUM_THREADS] = 1;
  printf("P%d : put F%d\n", i, (i+1)%NUM_THREADS);
  pthread_cond_signal(&conditionVars[(i+1)%NUM_THREADS]);
  pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);
```

Example of Dining Philosophers

```
// putdown left fork
 pthread_mutex_lock(&mutexes[i%NUM_THREADS]);
 permits[i%NUM_THREADS] = 1;
 printf("P%d : put F%d \n", i, i%NUM_THREADS);
 pthread_cond_signal(&conditionVars[i%NUM_THREADS]);
 pthread_mutex_unlock(&mutexes[i%NUM_THREADS]);

 // putdown right fork
 pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
 permits[(i+1)%NUM_THREADS] = 1;
 printf("P%d : put F%d \n", i, (i+1)%NUM_THREADS);
 pthread_cond_signal(&conditionVars[(i+1)%NUM_THREADS]);
 pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);


 return NULL;

}

int main(){
 int i;

 for (i = 0; i < NUM_THREADS; i++)
   pthread_mutex_init(&mutexes[i], NULL);
 for (i = 0; i < NUM_THREADS; i++)
   pthread_cond_init(&conditionVars[i], NULL);
 for (i = 0; i < NUM_THREADS; i++)
   permits[i] = 1;

 for (i = 0; i < NUM_THREADS-1; i++){
   pthread_create(&tids[i], NULL,  Philosopher, (void*)(i) );
 }

 pthread_create(&tids[NUM_THREADS-1], NULL,
   OddPhilosopher, (void*)(NUM_THREADS-1) );

 for (i = 0; i < NUM_THREADS; i++){
   pthread_join(tids[i], NULL);
 }

 for (i = 0; i < NUM_THREADS; i++){
   pthread_mutex_destroy(&mutexes[i]);
 }
 for (i = 0; i < NUM_THREADS; i++){
   pthread_cond_destroy(&conditionVars[i]);
 }

 //printf(" data = %d \n", data);

 //assert( data != 201);
 return 0;
}
```



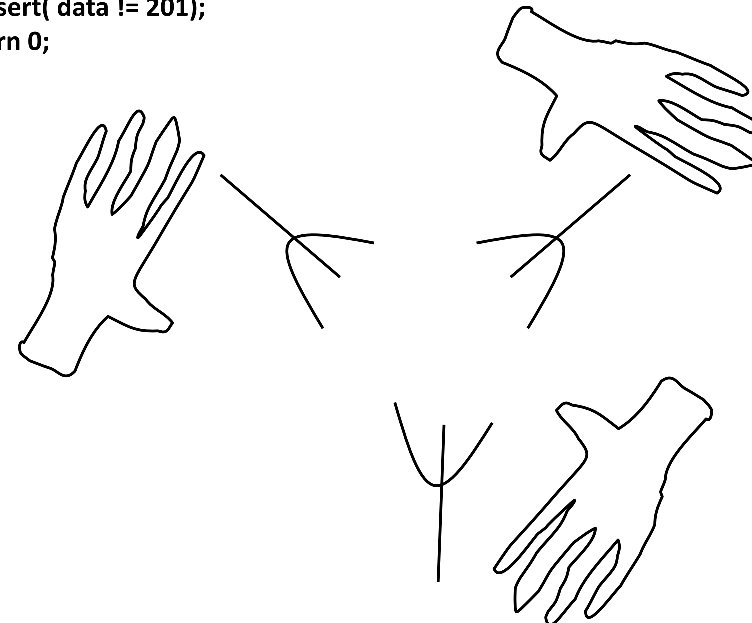
‘Plain run’ of Philosophers

```
gcc -g -O3 -o nobug examples/Dining3.c  -L ./lib -lpthread -lstdc++ -lssl

% time nobug

P0 : get F0
P0 : get F1
0
P0 : put F1
P0 : put F0
P1 : get F1
P1 : get F2
1
P1 : put F2
P1 : put F1
P2 : get F0
P2 : get F2
2
P2 : put F2
P2 : put F0

real        0m0.075s
user        0m0.001s
sys         0m0.008s
```

Buggy Philosophers in Pthreads (see the hands below!)

```
// putdown left fork
 pthread_mutex_lock(&mutexes[i%NUM_THREADS]);
 permits[i%NUM_THREADS] = 1;
 printf("P%d : put F%d \n", i, i%NUM_THREADS);
 pthread_cond_signal(&conditionVars[i%NUM_THREADS]);
 pthread_mutex_unlock(&mutexes[i%NUM_THREADS]);

 // putdown right fork
 pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
 permits[(i+1)%NUM_THREADS] = 1;
 printf("P%d : put F%d \n", i, (i+1)%NUM_THREADS);
 pthread_cond_signal(&conditionVars[(i+1)%NUM_THREADS]);
 pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);


 return NULL;

}

int main(){
 int i;

 for (i = 0; i < NUM_THREADS; i++)
  pthread_mutex_init(&mutexes[i], NULL);
 for (i = 0; i < NUM_THREADS; i++)
  pthread_cond_init(&conditionVars[i], NULL);
 for (i = 0; i < NUM_THREADS; i++)
  permits[i] = 1;

 for (i = 0; i < NUM_THREADS-1; i++){
  pthread_create(&tids[i], NULL,  Philosopher, (void*)(i) );
 }
```

```
pthread_create(&tids[NUM_THREADS-1], NULL,
  Philosopher, (void*)(NUM_THREADS-1) );

 for (i = 0; i < NUM_THREADS; i++){
  pthread_join(tids[i], NULL);
 }

 for (i = 0; i < NUM_THREADS; i++){
  pthread_mutex_destroy(&mutexes[i]);
 }
 for (i = 0; i < NUM_THREADS; i++){
  pthread_cond_destroy(&conditionVars[i]);
 }

 //printf(" data = %d \n", data);

 //assert( data != 201);
 return 0;
}
```



‘Plain run’ of buggy philosopher .. bugs missed by testing

```
gcc -g -O3 -o buggy examples/Dining3Buggy.c  -L ./lib -lpthread -lstdc++ -lssl

% time buggy

P0 : get F0
P0 : get F1
0
P0 : put F1
P0 : put F0
P1 : get F1
P1 : get F2
1
P1 : put F2
P1 : put F1
P2 : get F2
P2 : get F0
2
P2 : put F0
P2 : put F2

real        0m0.084s
user        0m0.002s
sys         0m0.011s
```

Jiggling Schedule in Buggy Philosopher.. Didn't help!

```
#include <stdlib.h>    // Dining Philosophers with no deadlock
#include <pthread.h>   // all phils but "odd" one pickup their
#include <stdio.h>     // left fork first; odd phil picks
#include <string.h>    // up right fork first
#include <malloc.h>
#include <errno.h>
#include <sys/types.h>
#include <assert.h>

#define NUM_THREADS 3

pthread_mutex_t mutexes[NUM_THREADS];
pthread_cond_t conditionVars[NUM_THREADS];
int permits[NUM_THREADS];
pthread_t tids[NUM_THREADS];

int data = 0;

void * Philosopher(void * arg){
  int i;
  i = (int)arg;

  // pickup left fork
  pthread_mutex_lock(&mutexes[i%NUM_THREADS]);
  while (permits[i%NUM_THREADS] == 0) {
    printf("P%d : tryget F%d\n", i, i%NUM_THREADS);
    pthread_cond_wait(&conditionVars[i%NUM_THREADS],&mutexes[i
%NUM_THREADS]);
  }
permits[i%NUM_THREADS] = 0;
  printf("P%d : get F%d\n", i, i%NUM_THREADS);
  pthread_mutex_unlock(&mutexes[i%NUM_THREADS]);

   nanosleep (0) added here

  // pickup right fork
  pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
  while (permits[(i+1)%NUM_THREADS] == 0) {
    printf("P%d : tryget F%d\n", i, (i+1)%NUM_THREADS);
    pthread_cond_wait(&conditionVars[(i+1)%NUM_THREADS],&mutexes[(i
+1)%NUM_THREADS]);
 }
  permits[(i+1)%NUM_THREADS] = 0;
  printf("P%d : get F%d\n", i, (i+1)%NUM_THREADS);
  pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);

  //printf("philosopher %d thinks \n",i);
  printf("%d\n", i);

  //  data = 10 * data + i;

  fflush(stdout);

  // putdown right fork
  pthread_mutex_lock(&mutexes[(i+1)%NUM_THREADS]);
  permits[(i+1)%NUM_THREADS] = 1;
  printf("P%d : put F%d\n", i, (i+1)%NUM_THREADS);
  pthread_cond_signal(&conditionVars[(i+1)%NUM_THREADS]);
  pthread_mutex_unlock(&mutexes[(i+1)%NUM_THREADS]);
```

'Plain runs' of buggy philosopher – bug still very dodgy …

```
gcc -g -O3 -o
buggynsleep examples/Dining3BuggyNanosleep0.c
  -L ./lib -lpthread -lstdc++ -lssl

% buggysleep

P0 : get F0
P0 : sleeping 0 ns
P1 : get F1
P1 : sleeping 0 ns
P2 : get F2
P2 : sleeping 0 ns
P0 : tryget F1
P2 : tryget F0
P1 : tryget F2
```

```
buggysleep

P0 : get F0
P0 : sleeping 0 ns
P0 : get F1
0
P0 : put F1
P0 : put F0
P1 : get F1
P1 : sleeping 0 ns
P2 : get F2
P2 : sleeping 0 ns
P1 : tryget F2
P2 : get F0
2
P2 : put F0
P2 : put F2
P1 : get F2
1
P1 : put F2
P1 : put F1
```

First run deadlocked – second did not ..

Inspect of nonbuggy and buggy Philosophers ..

```
./instrument file.c
./compile file.instr.c
./inspect ./target

P0 : get F0
P0 : get F1
0
P0 : put F1
P0 : put F0
P1 : get F1
P1 : get F2
1
P1 : put F2
P1 : put F1
P2 : get F0
P2 : get F2
2
P2 : put F2
P2 : put F0
 num of threads = 1
 === run 2 ===
P0 : get F0
…
P1 : put F1
```

```
…
=== run 48 ===
P2 : get F0
P2 : get F2
2
P2 : put F2
P2 : put F0
P0 : get F0
P0 : get F1
0
P1 : tryget F1
<<
Total number
of runs:
48,
Transitions
explored: 1814
Used time
(seconds): 7.999327
```

```
 === run 1 ===
P0 : get F0
P0 : get F1
0
P0 : put F1
P0 : put F0
P1 : get F1
P1 : get F2
1
P1 : put F2
P1 : put F1
P2 : get F2
P2 : get F0
2
P2 : put F0
P2 : put F2
 === run 2 ===
P0 : get F0
P0 : get F1
0
P0 : put F1
P0 : put F0
P1 : get F1
P1 : get F2
1
P2 : tryget F2
P1 : put F2
P1 : put F1
```

```
 === run 28 ===
P0 : get F0
P1 : get F1
P0 : tryget F1
P2 : get F2
P1 : tryget F2
P2 : tryget F0
Found a deadlock!!
(0, thread_start)
(0, mutex_init, 5)
(0, mutex_init, 6)
(0, mutex_init, 7)
(0, cond_init, 8)
(0, cond_init, 9)
(0, cond_init, 10)
(0, obj_write, 2)
(0, obj_write, 3)
(0, obj_write, 4)
(0, thread_create, 1)
(0, thread_create, 2)
(0, thread_create, 3)
(1, mutex_lock, 5)
(1, obj_read, 2)
(1, obj_write, 2)
(1, mutex_unlock, 5)
(2, mutex_lock, 6)
(2, obj_read, 3)
```

```
(2, obj_write, 3)
(2, mutex_unlock, 6)
(1, mutex_lock, 6)
(1, obj_read, 3)
(1, mutex_unlock, 6)
(3, mutex_lock, 7)
(3, obj_read, 4)
(3, obj_write, 4)
(3, mutex_unlock, 7)
(2, mutex_lock, 7)
(2, obj_read, 4)
(2, mutex_unlock, 7)
(3, mutex_lock, 5)
(3, obj_read, 2)
(3, mutex_unlock, 5)
(-1, unknown)

Total number of runs:
29,
killed-in-the-middle runs:
4
Transitions explored:
1193
Used time (seconds):
5.990523
```

The Growth of $(n.p)! / (n!)^p$ for Diningp.c

- **Diningp.c has n = 4 (roughly)**
- **p = 3 : We get 34,650 (loose upper-bound) versus 48 with DPOR**
- **p = 5 : We get 305,540,235,000 versus 2,375 with DPOR**
- **DPOR really works well in reducing the number of interleavings !!**
- **Testing will have to exhibit its cleverness among $3 * 10^{11}$ interleavings**

HUGE importance of DPOR : After instrumentation the code expands, further increasing the number of interleavings!

BEFORE INSTRUMENTATION

```
void * thread_A(void* arg)
{
  pthread_mutex_lock(&mutex);
  A_count++;
  pthread_mutex_unlock(&mutex);
}


void * thread_B(void * arg)
{
  pthread_mutex_lock(&lock);
  B_count++;
  pthread_mutex_unlock(&lock);
}
```

AFTER INSTRUMENTATION (transitions are shown as bands)

```
void *thread_A(void *arg ) // thread_B is  similar
{ void *__retres2 ;
  int __cil_tmp3 ;
  int __cil_tmp4 ;

  {
  inspect_thread_start("thread_A");
  inspect_mutex_lock(& mutex);
  __cil_tmp4 = read_shared_0(& A_count);
  __cil_tmp3 = __cil_tmp4 + 1;
  write_shared_1(& A_count, __cil_tmp3);
  inspect_mutex_unlock(& mutex);
  __retres2 = (void *)0;
  inspect_thread_end();
  return (__retres2);
}
}
```

HUGE importance of DPOR : After instrumentation the code expands, further increasing the number of interleavings!

BEFORE INSTRUMENTATION

```
void * thread_A(void* arg)
{
  pthread_mutex_lock(&mutex);
  A_count++;
  pthread_mutex_unlock(&mutex);
}


void * thread_B(void * arg)
{
  pthread_mutex_lock(&lock);
  B_count++;
  pthread_mutex_unlock(&lock);
}
```

AFTER INSTRUMENTATION
(transitions are shown as bands)

```
void *thread_A(void *arg ) // thread_B is  similar
{ void *__retres2 ;
  int __cil_tmp3 ;
  int __cil_tmp4 ;

  {
  inspect_thread_start("thread_A");
  inspect_mutex_lock(& mutex);
  __cil_tmp4 = read_shared_0(& A_count);
  __cil_tmp3 = __cil_tmp4 + 1;
  write_shared_1(& A_count, __cil_tmp3);
  inspect_mutex_unlock(& mutex);
  __retres2 = (void *)0;
  inspect_thread_end();
  return (__retres2);
}
}
```

"Look, ma, no dependencies!"

- **ONE interleaving with DPOR**
- **252 = (10!) / $(5!)^2$ without DPOR**

Examples Included With Tutorial

Dining3Buggy.c : Initial attempt to write 3 Dining Philosophers.
Since the code is symmetric, it has a deadlock. Testing misses it.

Dining3BuggyRace1.c: Initial attempt to tweak the code results in read / write race which Inspect finds (testing misses race + deadlock)

Dining3BuggyRace2.c: Another race is now exposed by Inspect

Dining3BuggyNoRace.c: All races removed. Now testing sometimes finds the deadlock. Inspect always finds it.

Dining3.c: This is the final bug-fixed version.

Dining5.c: Without DPOR, this should generate too many states. With DPOR, the number of states / transitions is far fewer.

sharedArrayRace.c: A shared array program with a race.

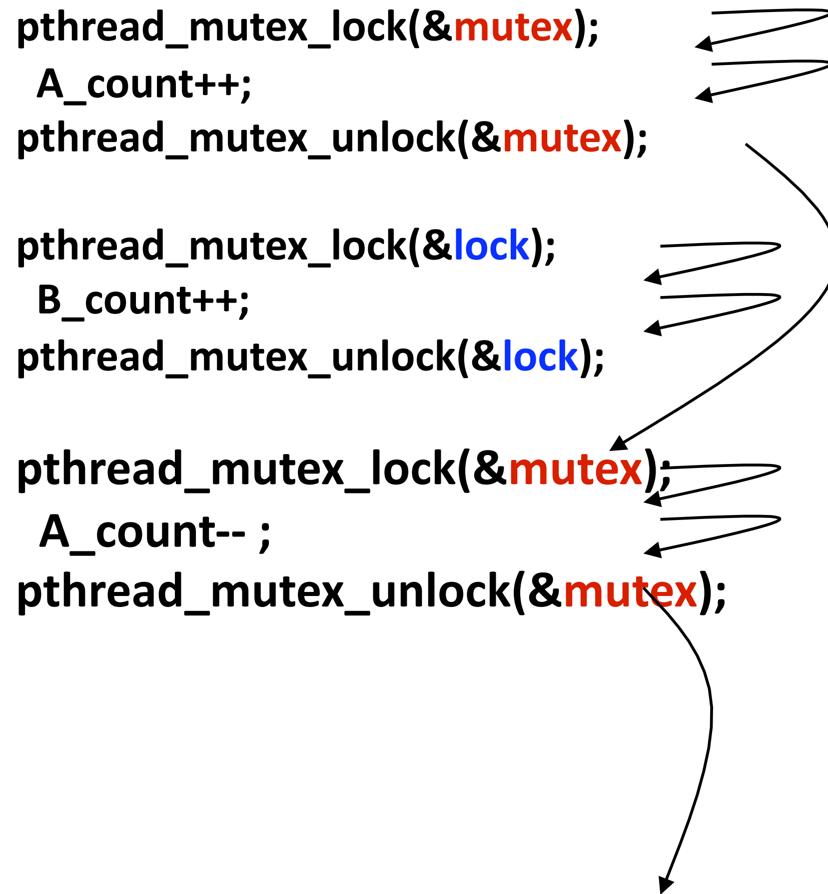
sharedArray.c: After fixing the race, stateless search does not finish.
We need stateful search to finish.

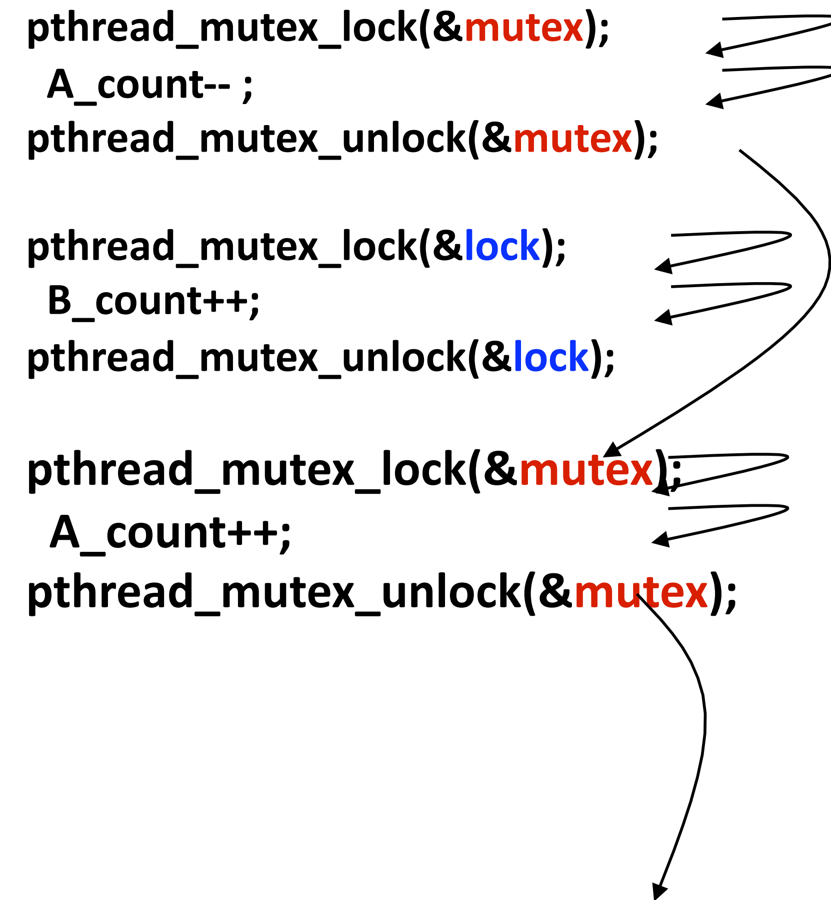
Happens-Before is defined by the Transition Dependency Relation

- Two transitions t1 and t2 of a concurrent program are **dependent**, if
  - **t1 and t2 belong to the same process, OR**
    - **t1 and t2 are concurrently enabled, and**

      t1, t2 are:
      - **lock acquire operations on the same lock**
      - **operations on the same global object and at least one of them is a write**
      - **a WAIT and a SIGNAL on the same condition variable**
- Introduce an HB edge between every pair of dependent operations in an execution

DPOR helps enumerate all possible "happens-before" partial orders…





Other details of DPOR

- Happens-before maintained using Vector Clocks

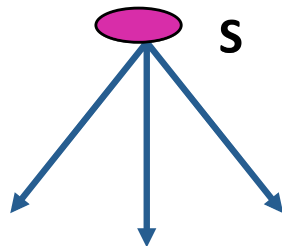
- Two transitions are concurrent if
  - They are not Happens-Before ordered
  - They can be executed under disjoint lock-sets

- DATA RACE
  - Two concurrent transitions enabled out of a state
  - Both access the same variable and one is a write

Computation of “ample” sets in Static POR versus in DPOR

Exploring “Ample” sets at every state suffices to generate all HB executions

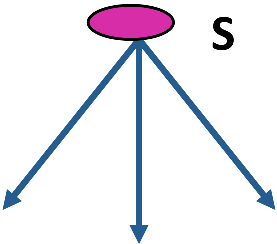
CLASSICAL POR :
AMPLE determined when S is reached

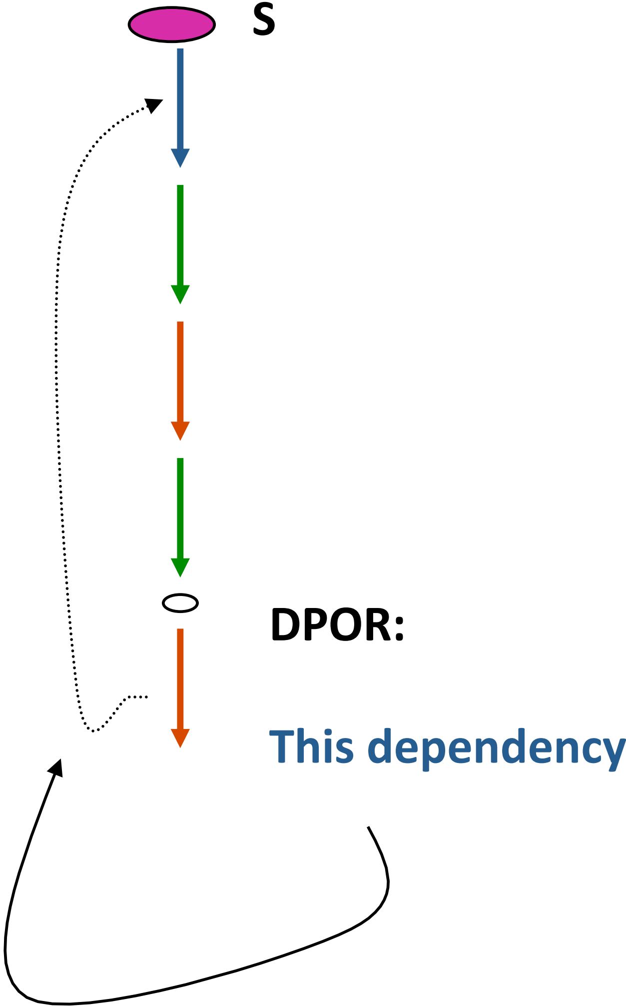


Computation of "ample" sets in Static POR versus in DPOR

Exploring "Ample" sets at every state suffices to generate all HB executions

CLASSICAL POR :
AMPLE determined when S is reached

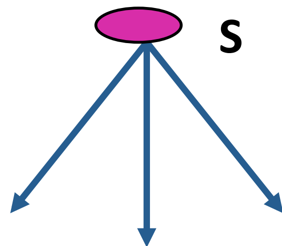


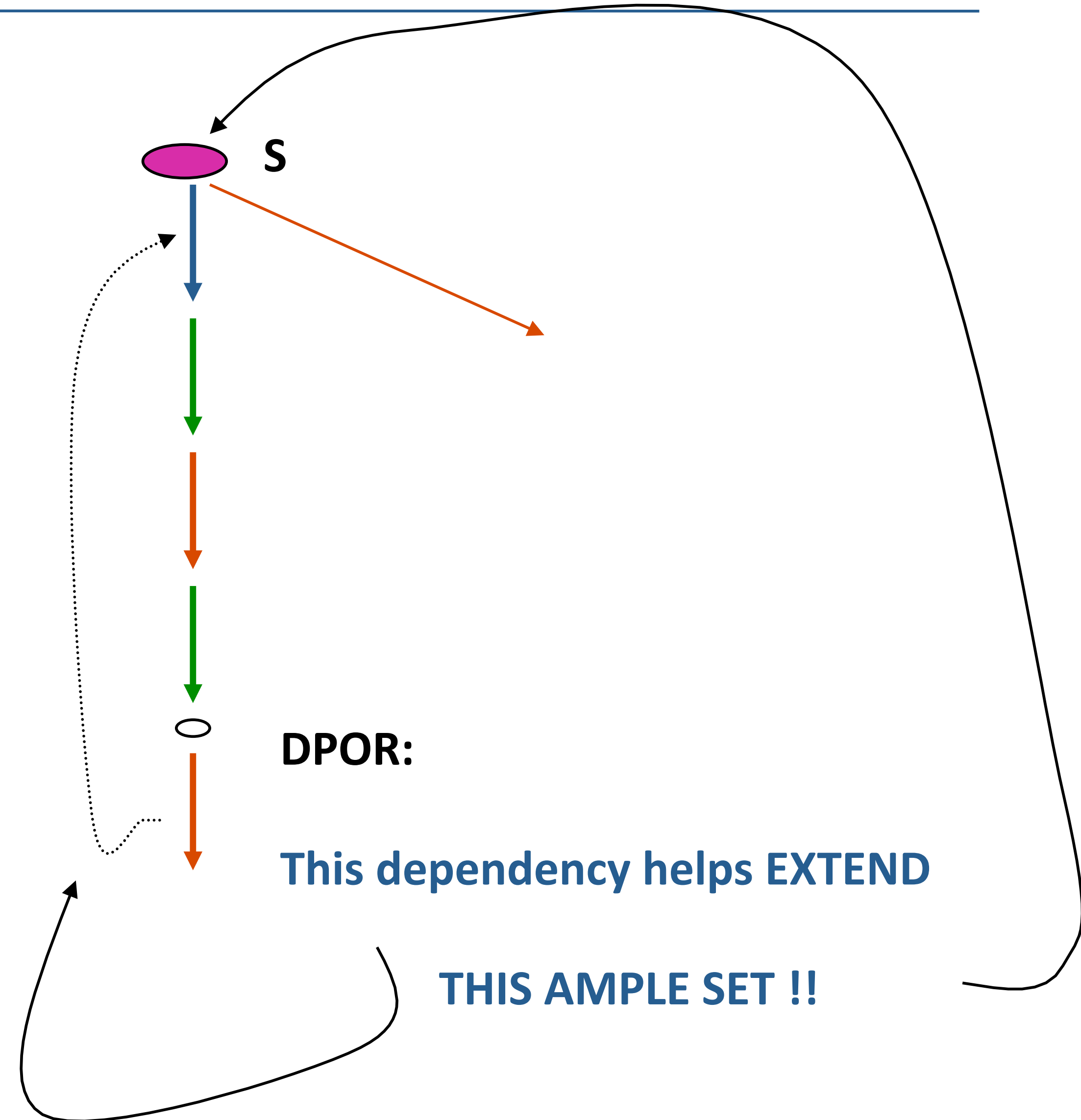


Computation of “ample” sets in Static POR versus in DPOR

Exploring “Ample” sets at every state suffices to generate all HB executions

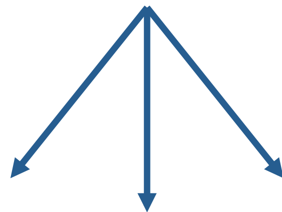
CLASSICAL POR :
AMPLE determined when S is reached





Computation of "ample" sets in Static POR versus in DPOR

Ample determined using "local" criteria



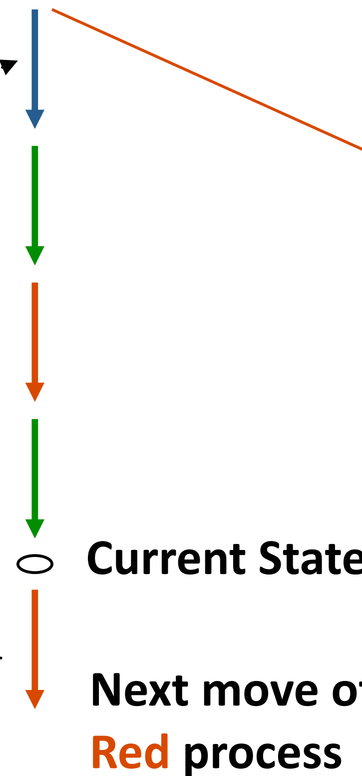
{ BT }, { Done }

Add Red Process to "Backtrack Set"

This builds the Ample set incrementally based on observed dependencies

Blue is in "Done" set

Nearest Dependent Transition Looking Back



Current State

Next move of Red process

Putting it all together …

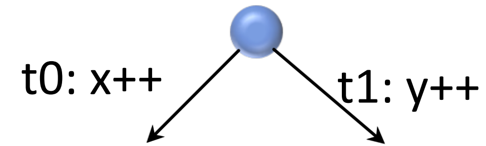
- We target C/C++ PThread Programs
- Instrument the given program (largely automated)
- Run the concurrent program "till the end"
- Compute dependencies based on concrete run information present in the runtime stack
  - This populates the Backtrack Sets -- points at which the execution must be replayed
- When an item (a process ID) is explored from the Backtrack Set, put it in the "done" set
- Repeat till all the Backtrack Sets are empty

A Simple DPOR Example

```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
     assert(false);

t1:
  y++;
  x++;
```

A Simple DPOR Example

{Backtrack}, { Done}

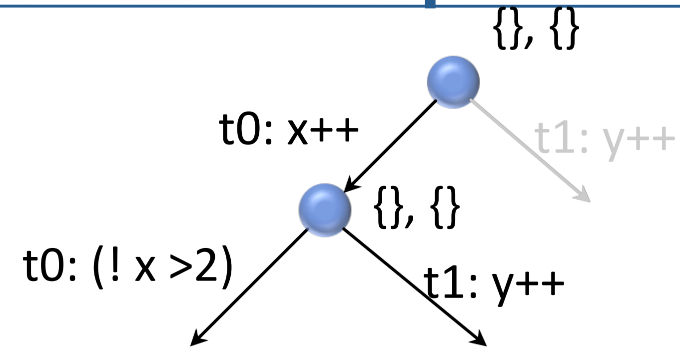



```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
     assert(false);

t1:
  y++;
  x++;
```

A Simple DPOR Example

{ Backtrack }, { Done}

```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
     assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, { Done}

init: x = 0; y = 0;

t0:

x++;

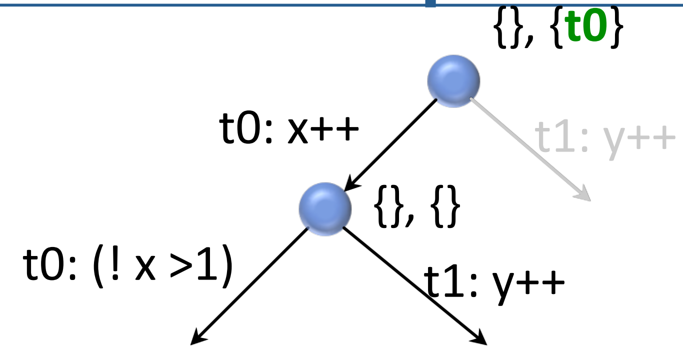
if (x > 1)

assert(false);

t1:

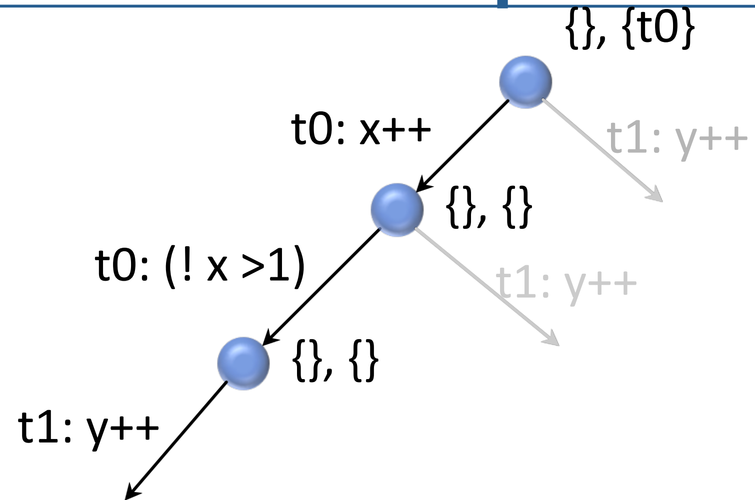
y++;

x++;



A Simple DPOR Example {Backtrack}, {Done}

```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
     assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack }, { Done }

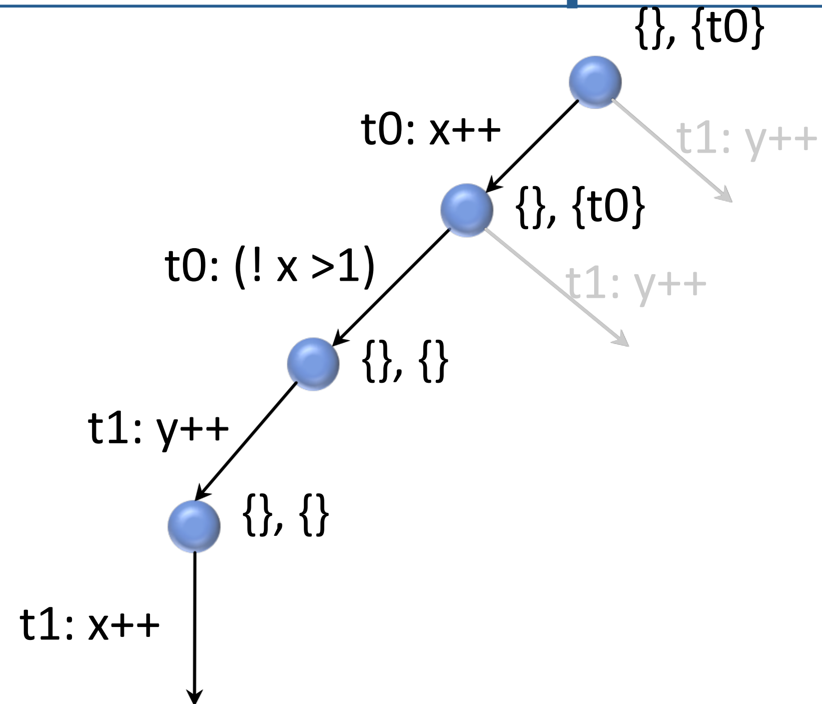
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```

{}, {t0}
t0: x++
t1: y++
{}, {t0}
t0: (! x >1)
t1: y++
{}, {}
t1: y++

A Simple DPOR Example

{Backtrack}, { Done }

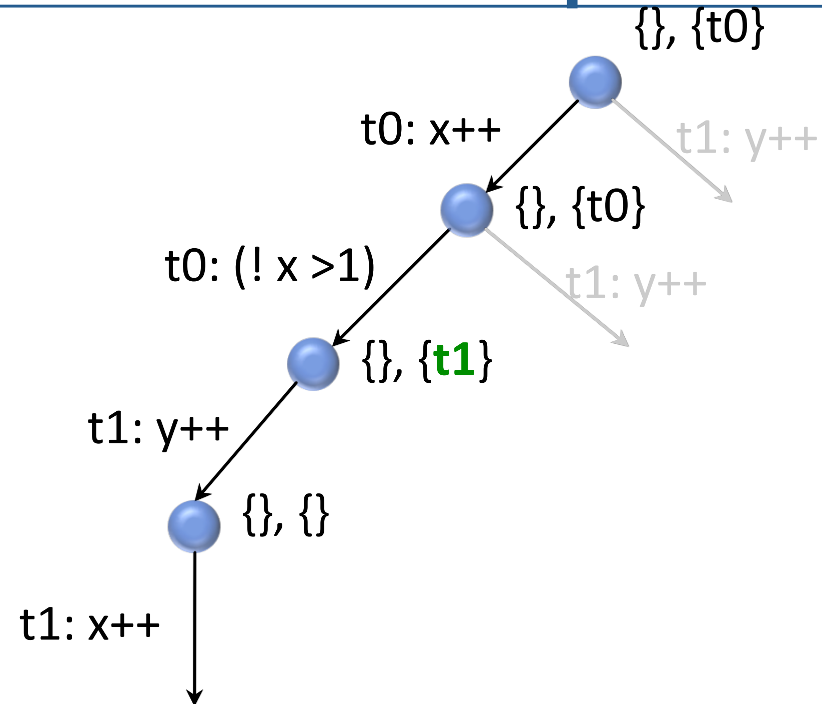
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, { Done }

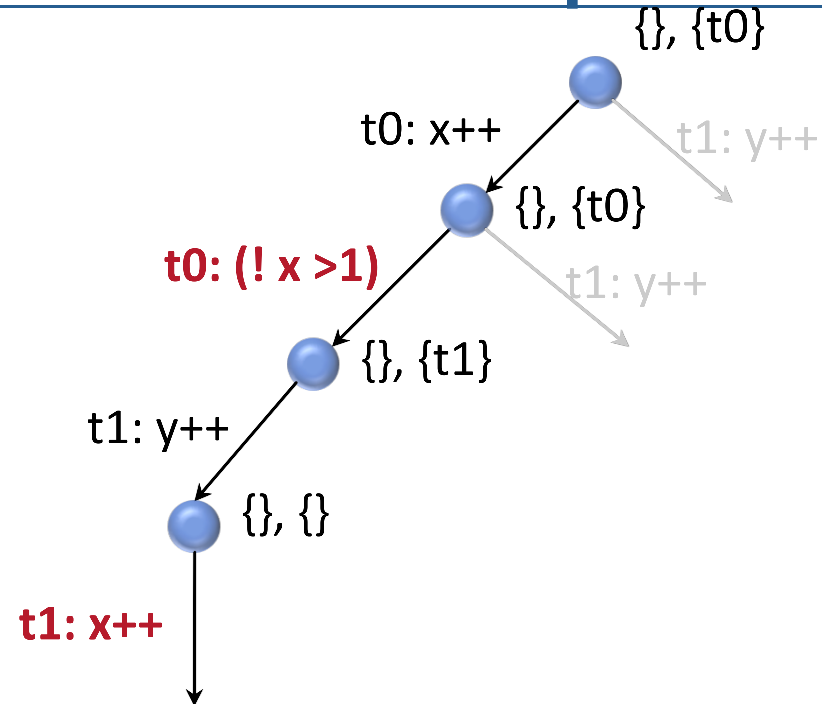
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, { Done }

```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```

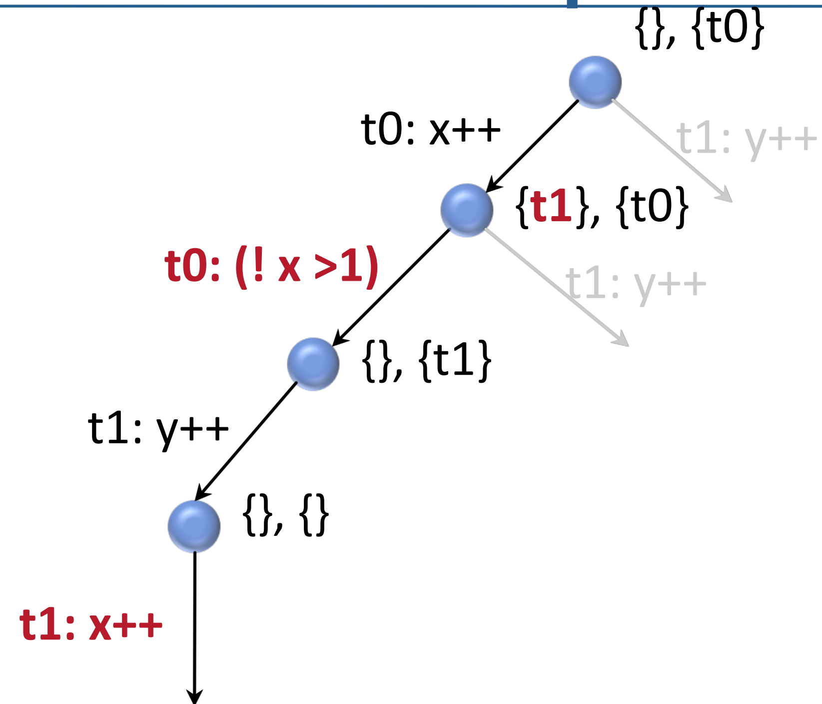


A Simple DPOR Example

{Backtrack}, { Done }

init: x = 0; y = 0;

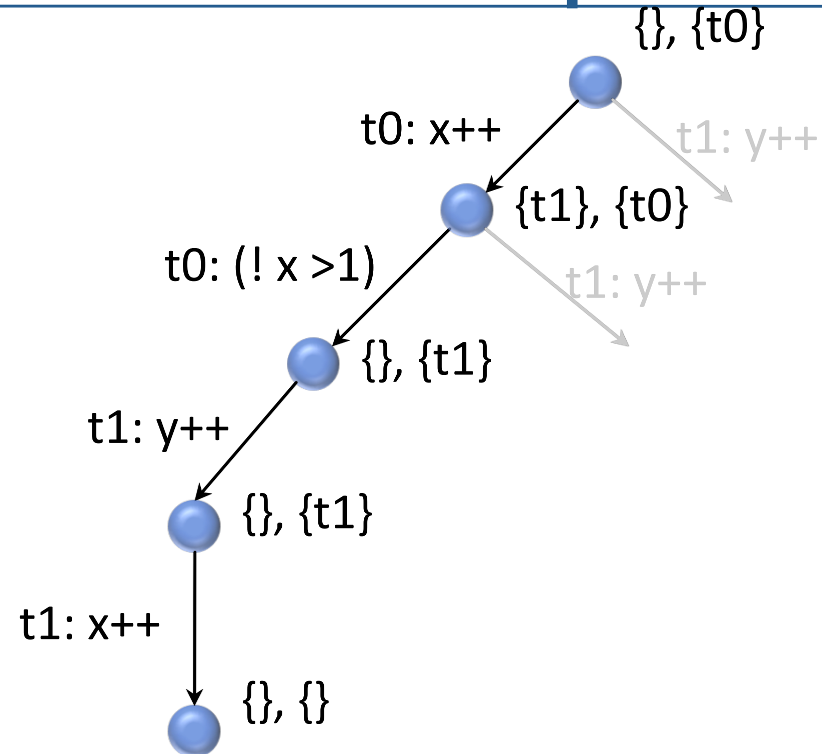
```
t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, { Done }

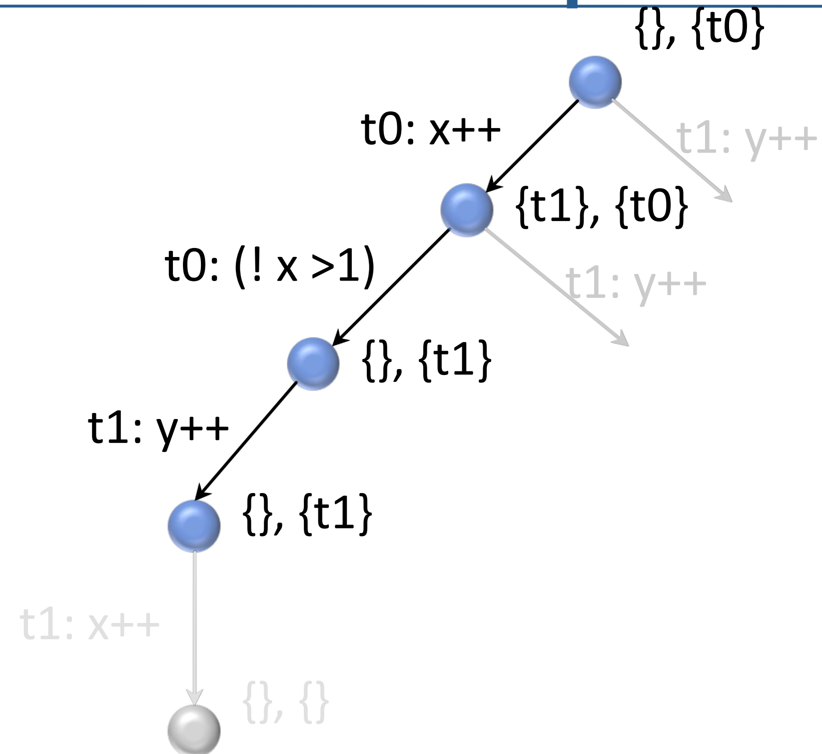
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, { Done }

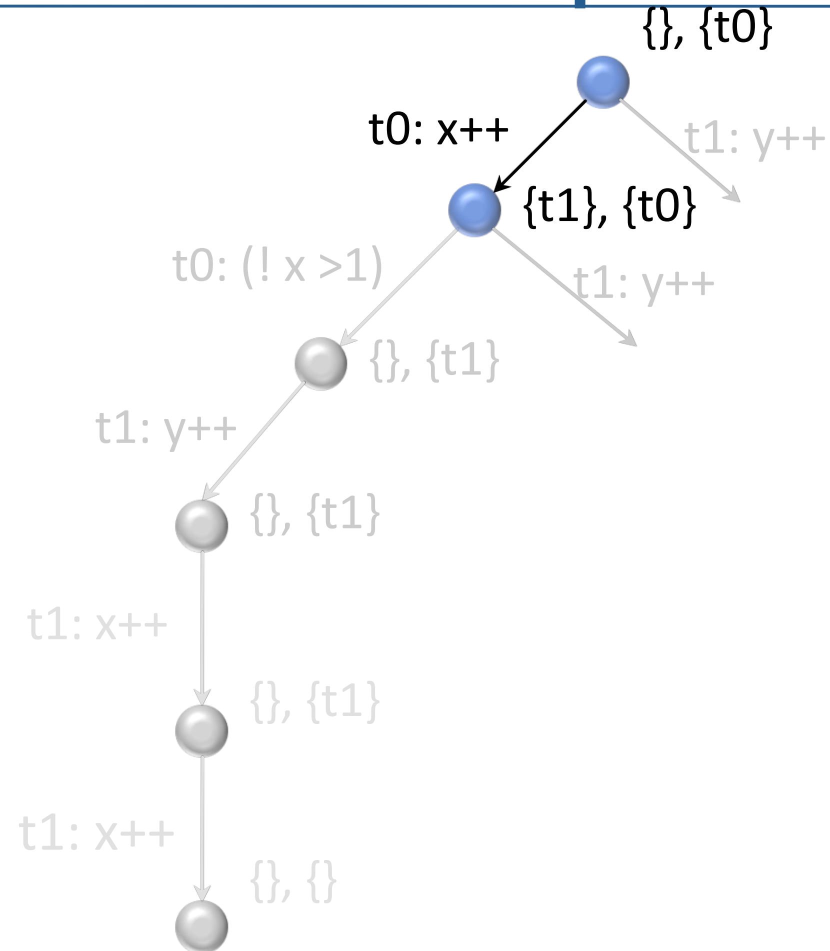
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, {Done}

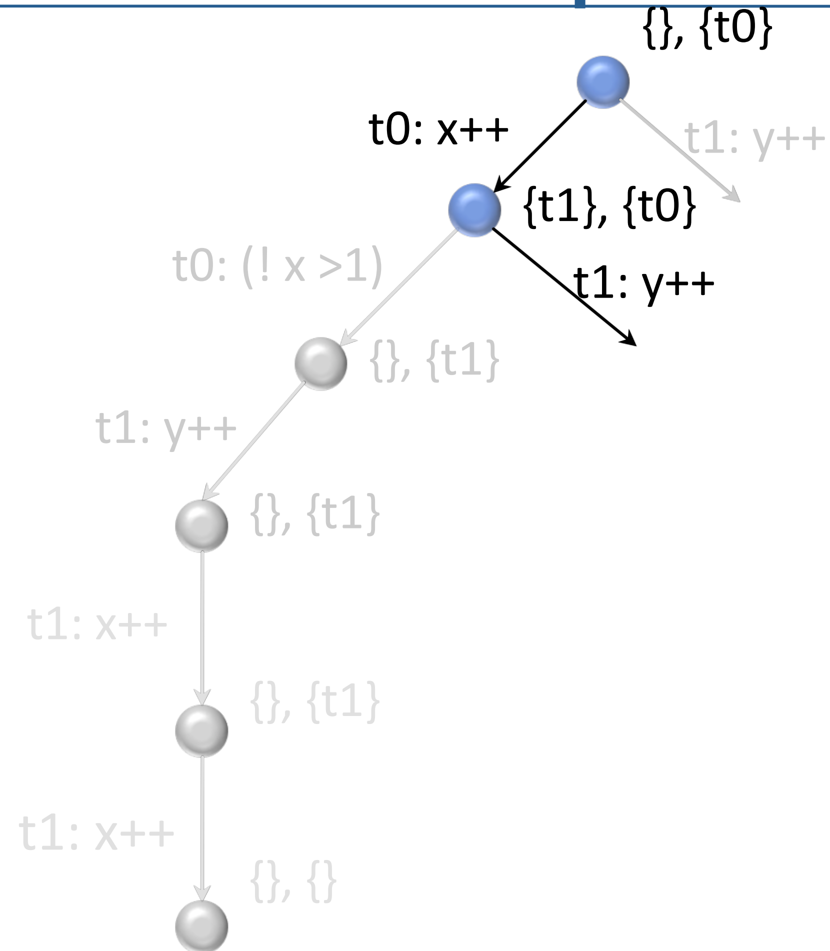
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, {Done}

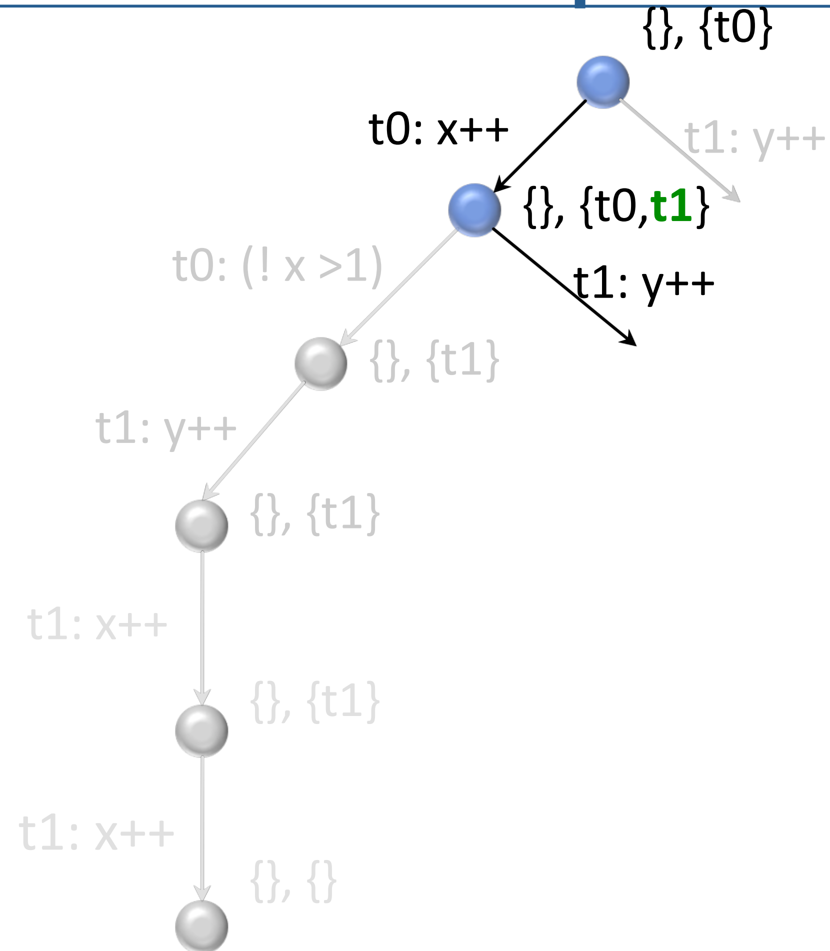
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, {Done}

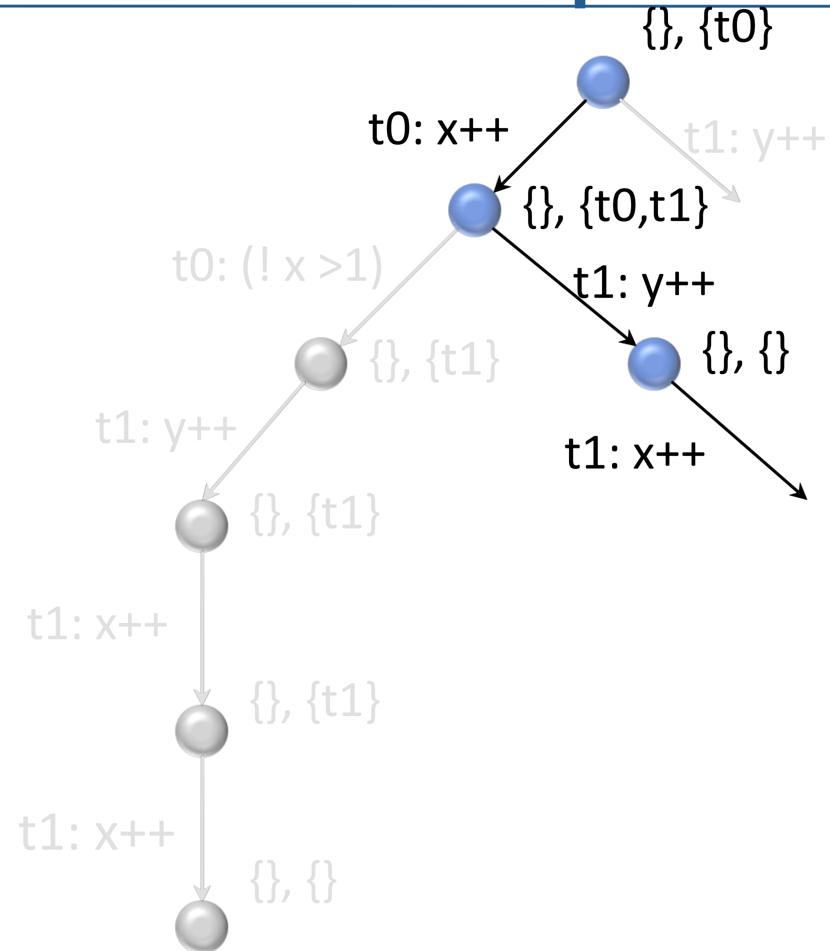
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, {Done}

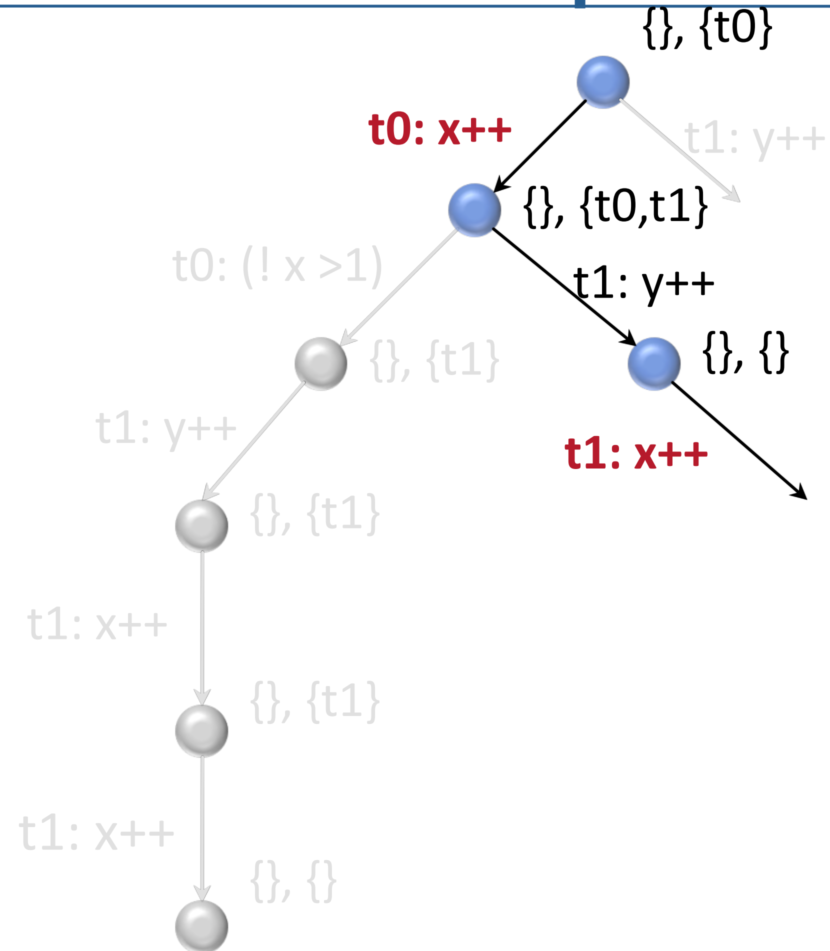
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, { Done }

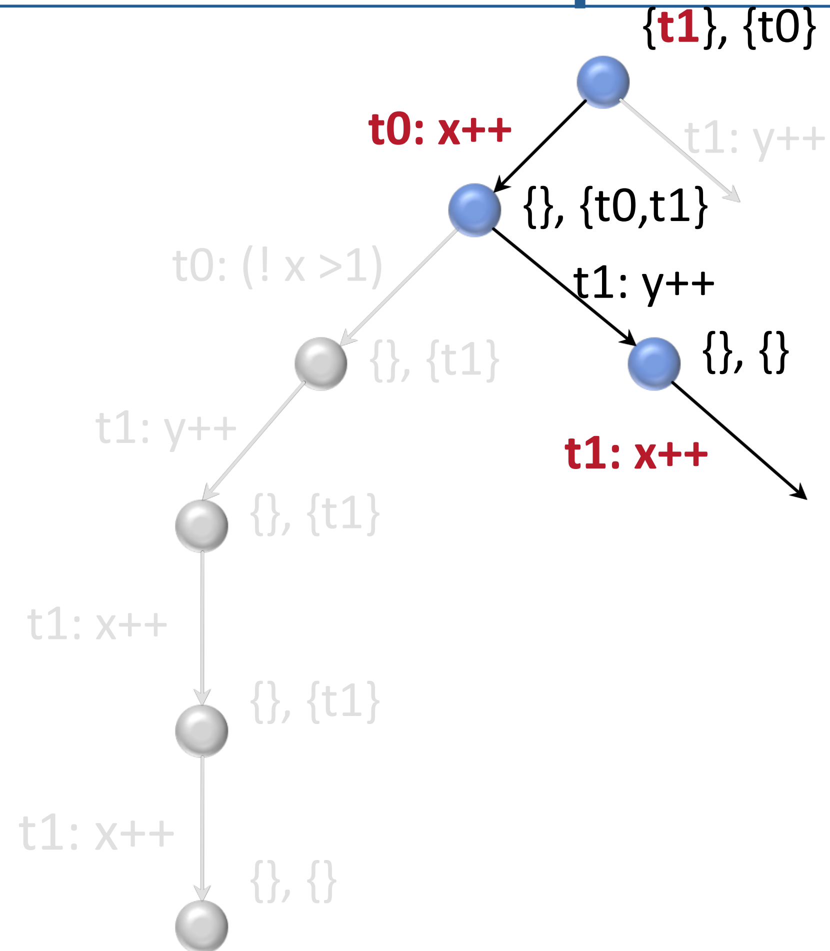
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, { Done }

```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, { Done }

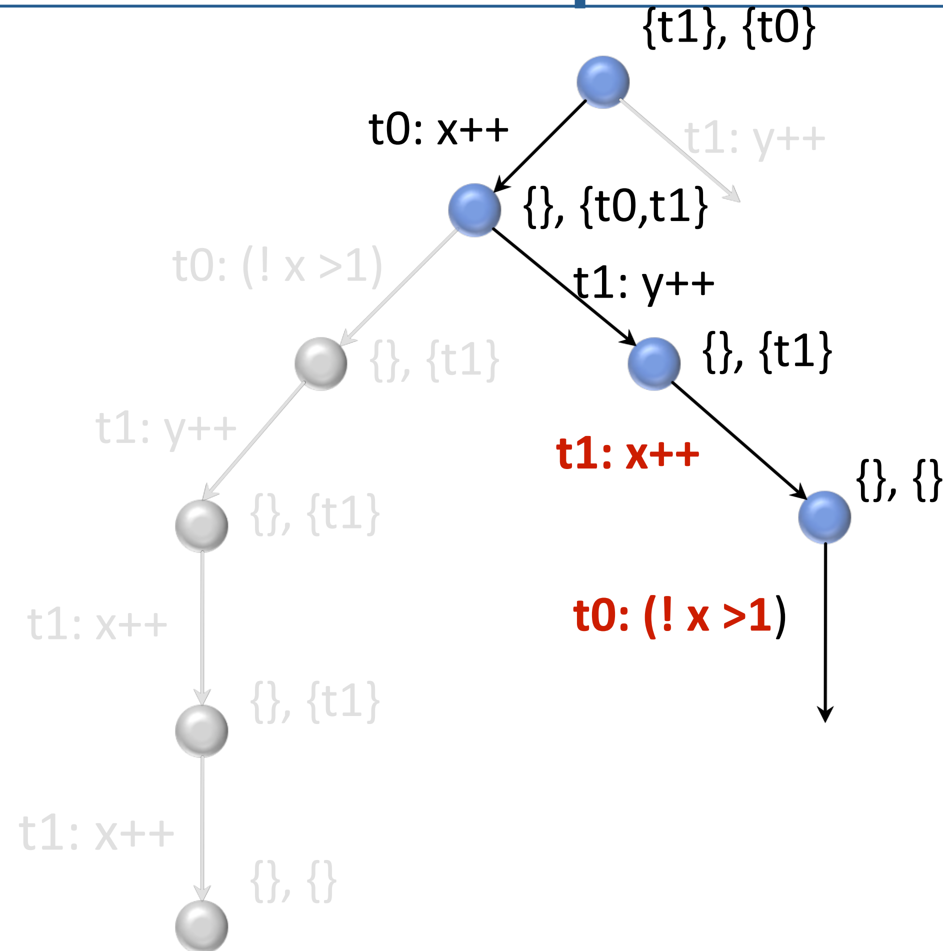
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```

{t1}, {t0}
t0: x++
t1: y++
{}, {t0,t1}
t0: (! x >1)
t1: y++
{}, {t1}
{}, {t1}
t1: y++
t1: x++
{}, {}
{}, {t1}
t1: x++
{}, {t1}
t0: (x >1)
t1: x++
{}, {}

A Simple DPOR Example

{Backtrack}, {Done}

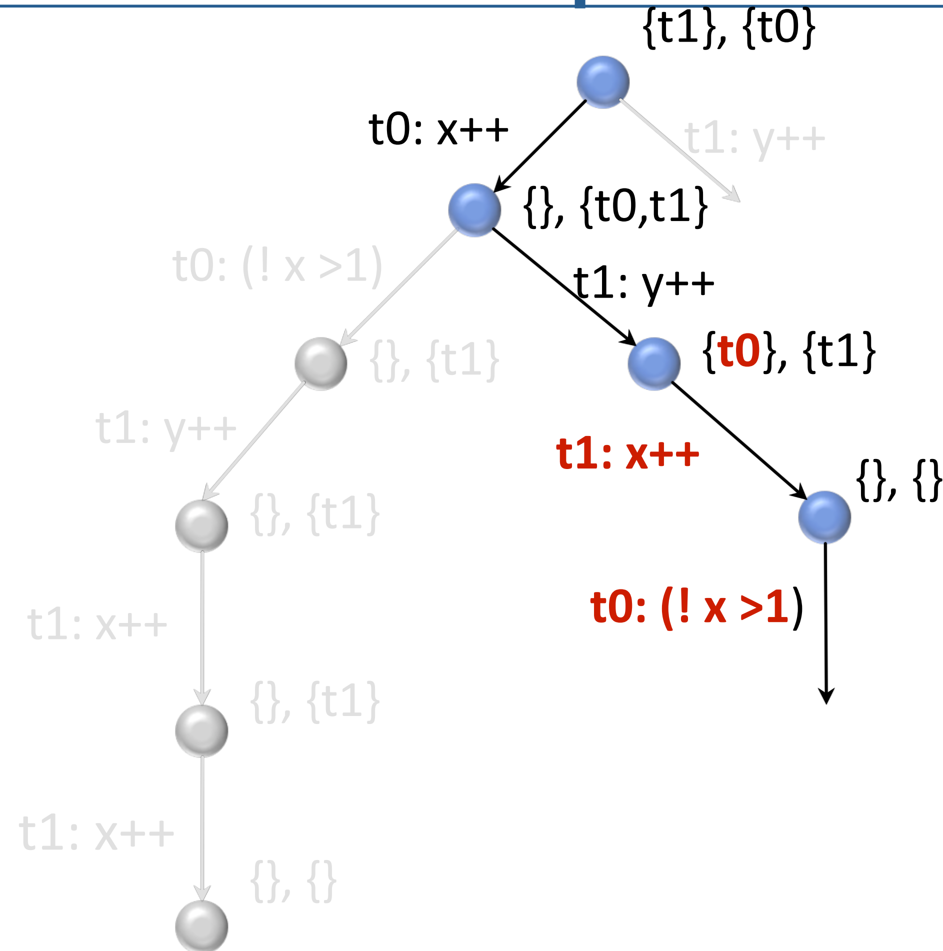
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, { Done }

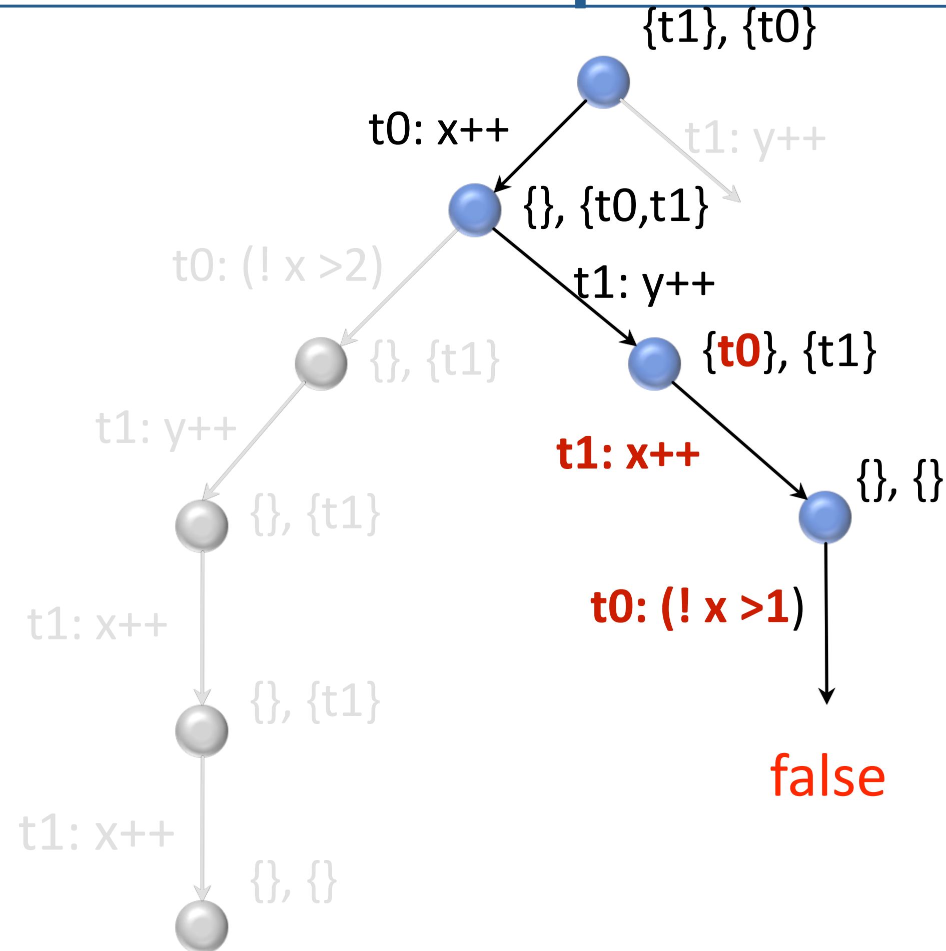
```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



A Simple DPOR Example

{Backtrack}, { Done }

```
init:  x = 0; y = 0;

t0:
  x++;
  if (x > 1)
    assert(false);

t1:
  y++;
  x++;
```



Evaluation

benchmark	LOC	thrds	DPOR			SDPOR		
			runs	trans	time(s)	runs	trans	time(s)
example1	40	2	-	-	-	35	2k	2
sharedArray	51	2	-	-	-	98	18k	6
bbuf	321	4	47K	1,058k	938	16k	350k	345
bzip2smp	6k	4	-	-	-	5k	26k	1311
bzip2smp	6k	5	-	-	-	18k	92k	9546
bzip2smp	6k	6	-	-	-	51k	236k	25659
pfscan	1k	3	84	1k	0.53	71	967	0.48
pfscan	1k	4	14k	189k	241	3k	40k	58
pfscan	1k	5	-	-	-	273k	3,402k	5329

Evaluation

benchmark	LOC	thrds	DPOR			SDPOR		
			runs	trans	time(s)	runs	trans	time(s)
example1	40	2	-	-	-	35	2k	2
sharedArray	51	2	-	-	-	98	18k	6
bbuf	321	4	47K	1,058k	938	16k	350k	345
bzip2smp	6k	4	-	-	-	5k	26k	1311
bzip2smp	6k	5	-	-	-	18k	92k	9546
bzip2smp	6k	6	-	-	-	51k	236k	25659
pfscan	1k	3	84	1k	0.53	71	967	0.48
pfscan	1k	4	14k	189k	241	3k	40k	58
pfscan	1k	5	-	-	-	273k	3,402k	5329

Evaluation

benchmark	LOC	thrds	DPOR			SDPOR		
			runs	trans	time(s)	runs	trans	time(s)
example1	40	2	-	-	-	35	2k	2
sharedArray	51	2	-	-	-	98	18k	6
bbuf	321	4	47K	1,058k	938	16k	350k	345
bzip2smp	6k	4	-	-	-	5k	26k	1311
bzip2smp	6k	5	-	-	-	18k	92k	9546
bzip2smp	6k	6	-	-	-	51k	236k	25659
pfscan	1k	3	84	1k	0.53	71	967	0.48
pfscan	1k	4	14k	189k	241	3k	40k	58
pfscan	1k	5	-	-	-	273k	3,402k	5329

{ BT }, { Done }

A Simple DPOR Example

{}, {}

```
t0:
  lock(t)
  unlock(t)

t1:
  lock(t)
  unlock(t)

t2:
  lock(t)
  unlock(t)
```

{ BT }, { Done }

A Simple DPOR Example

```
t0:
  lock(t)
  unlock(t)

t1:
  lock(t)
  unlock(t)

t2:
  lock(t)
  unlock(t)
```

{}, {}

t0: lock

{ BT }, { Done }

A Simple DPOR Example

t0:
  lock(t)
  unlock(t)

t1:
  lock(t)
  unlock(t)

t2:
  lock(t)
  unlock(t)

{}, {}

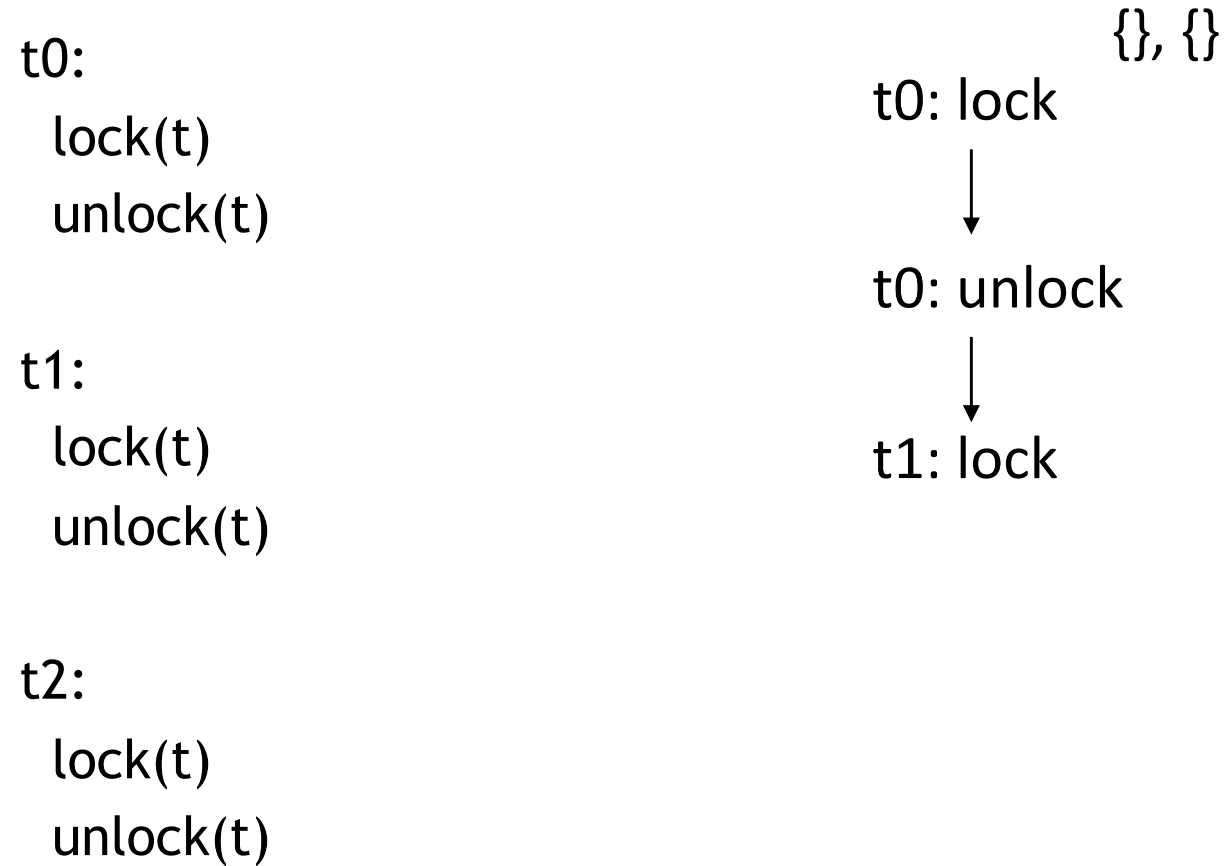
t0: lock

↓

t0: unlock

{ BT }, { Done }

A Simple DPOR Example



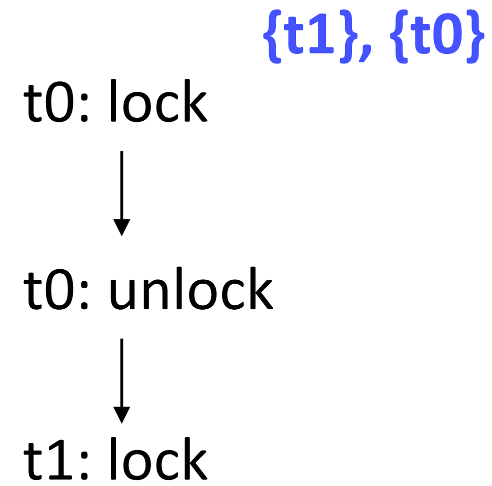
{ BT }, { Done }

A Simple DPOR Example

t0:
  lock(t)
  unlock(t)

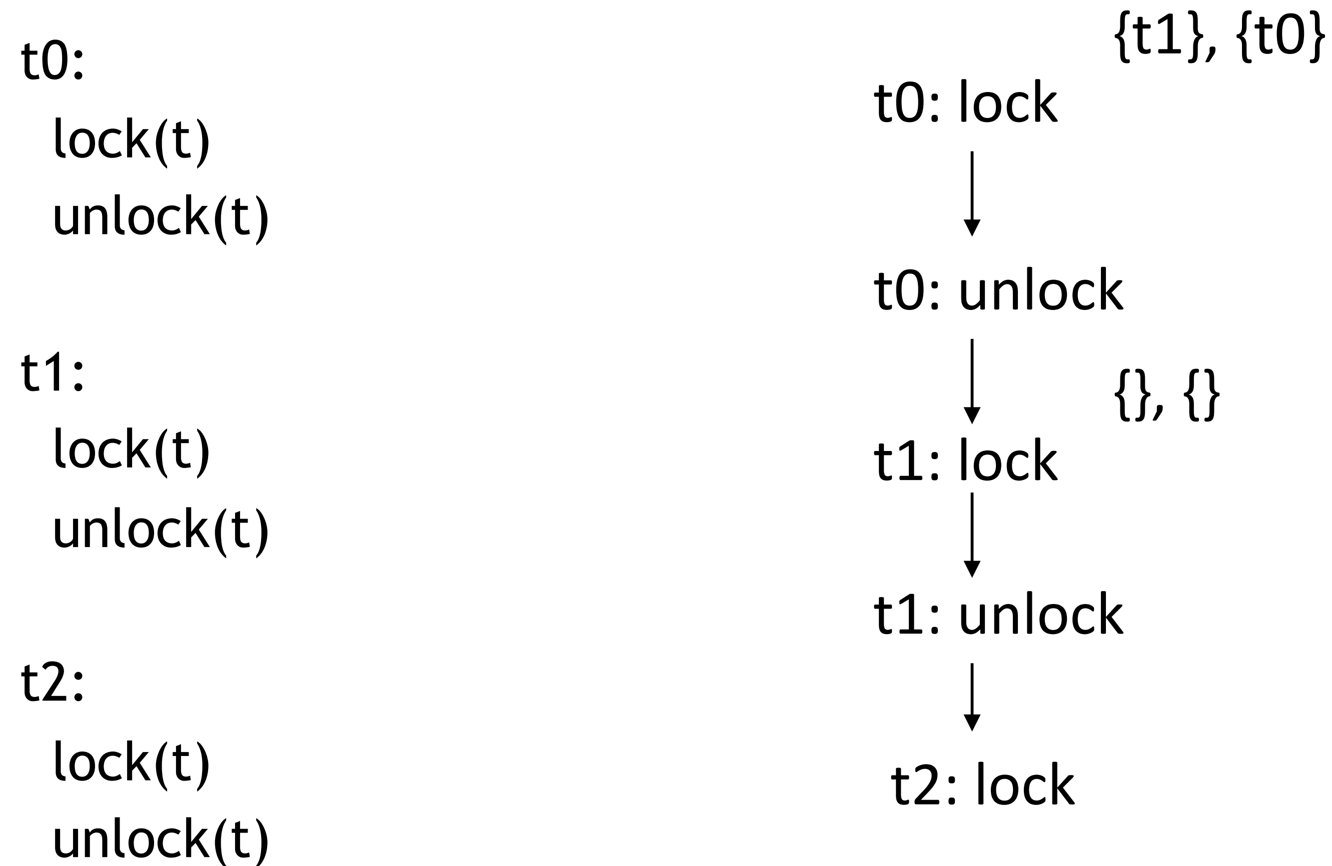
t1:
  lock(t)
  unlock(t)

t2:
  lock(t)
  unlock(t)



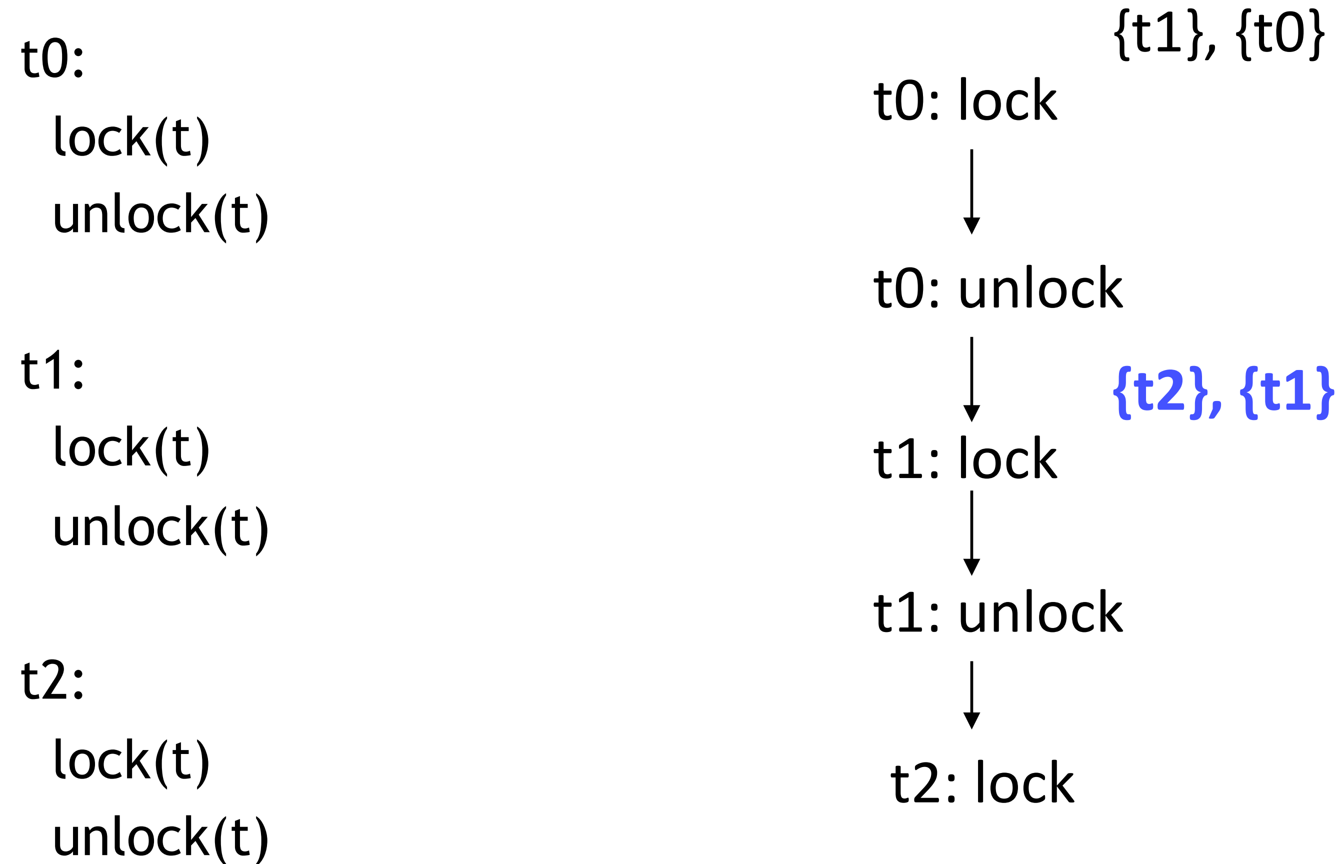
{ **BT** }, { **Done** }

A Simple DPOR Example



{ **BT** }, { **Done** }

A Simple DPOR Example



{ BT }, { Done }

A Simple DPOR Example

```
t0:
  lock(t)
  unlock(t)

t1:
  lock(t)
  unlock(t)

t2:
  lock(t)
  unlock(t)
```

{t1}, {t0}

t0: lock

↓

t0: unlock

↓

{t2}, {t1}

t1: lock

↓

t1: unlock

↓

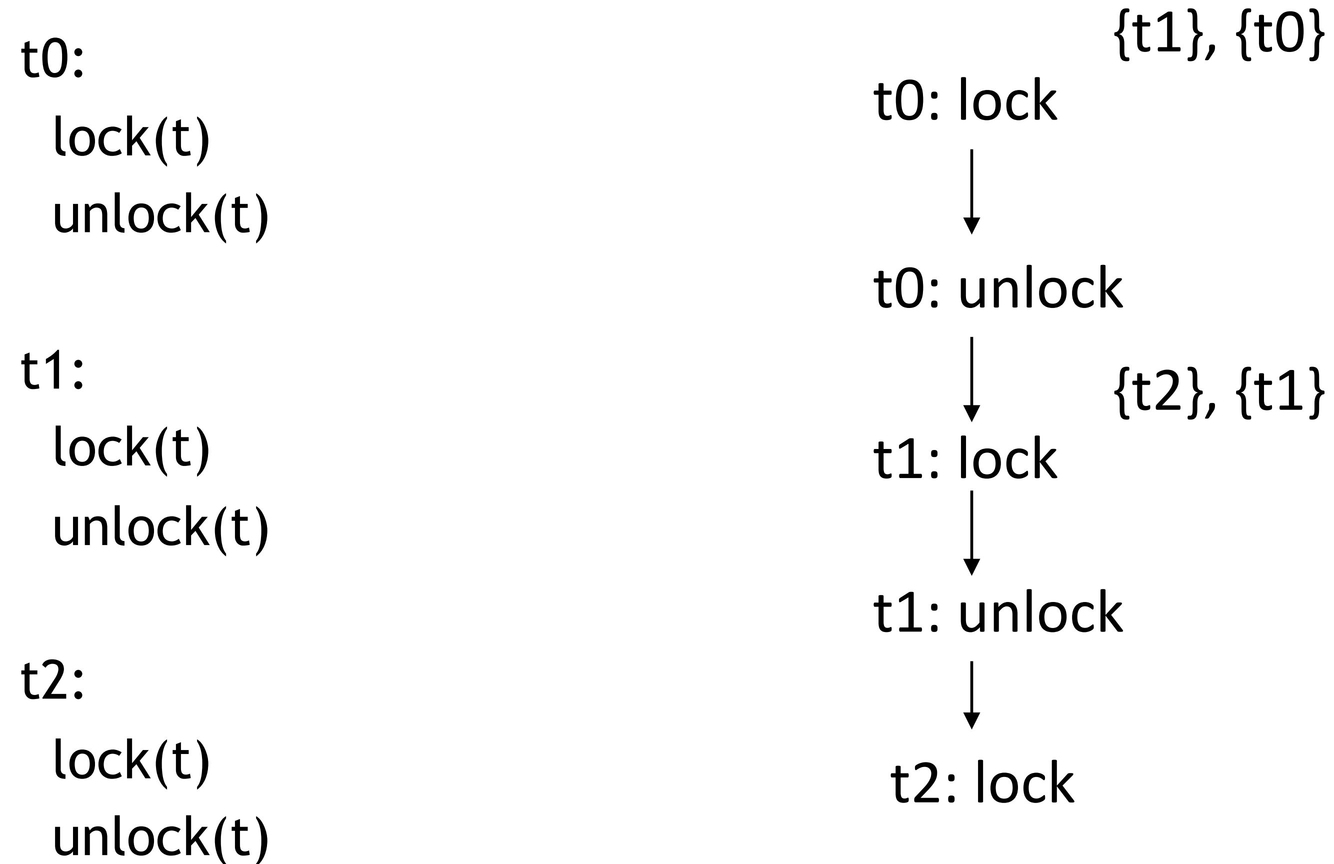
t2: lock

↓

t2: unlock

{ BT }, { Done }

A Simple DPOR Example



{ **BT** }, { **Done** }

A Simple DPOR Example

t0:
  lock(t)
  unlock(t)

t1:
  lock(t)
  unlock(t)

t2:
  lock(t)
  unlock(t)

{t1}, {t0}

t0: lock

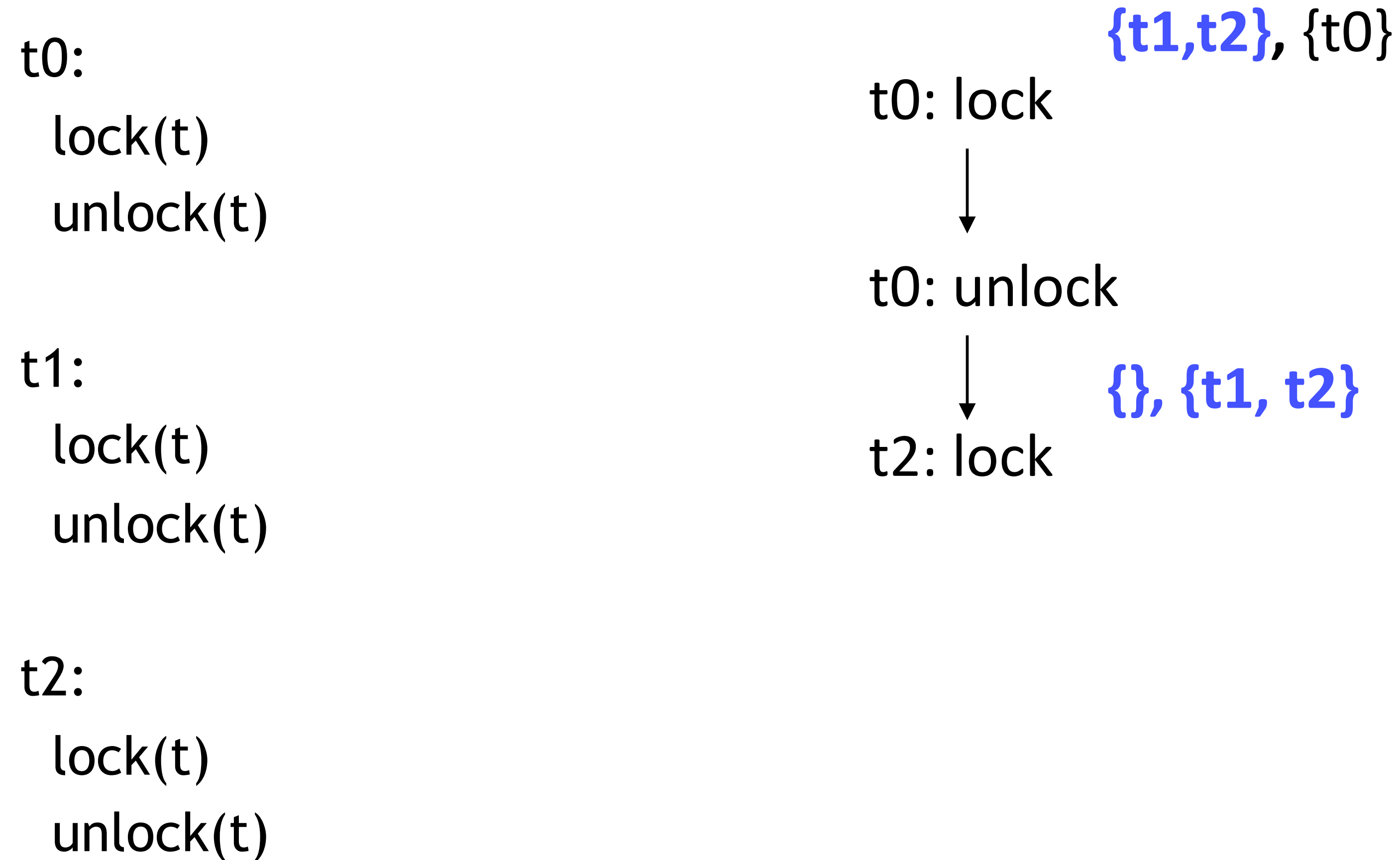
↓

t0: unlock

↓ {t2}, {t1}

{ **BT** }, { **Done** }

A Simple DPOR Example



{ BT }, { Done }

A Simple DPOR Example

```
t0:
  lock(t)
  unlock(t)

t1:
  lock(t)
  unlock(t)

t2:
  lock(t)
  unlock(t)
```

{t1,t2}, {t0}

t0: lock

↓

t0: unlock

↓ {}, {t1, t2}

t2: lock

↓

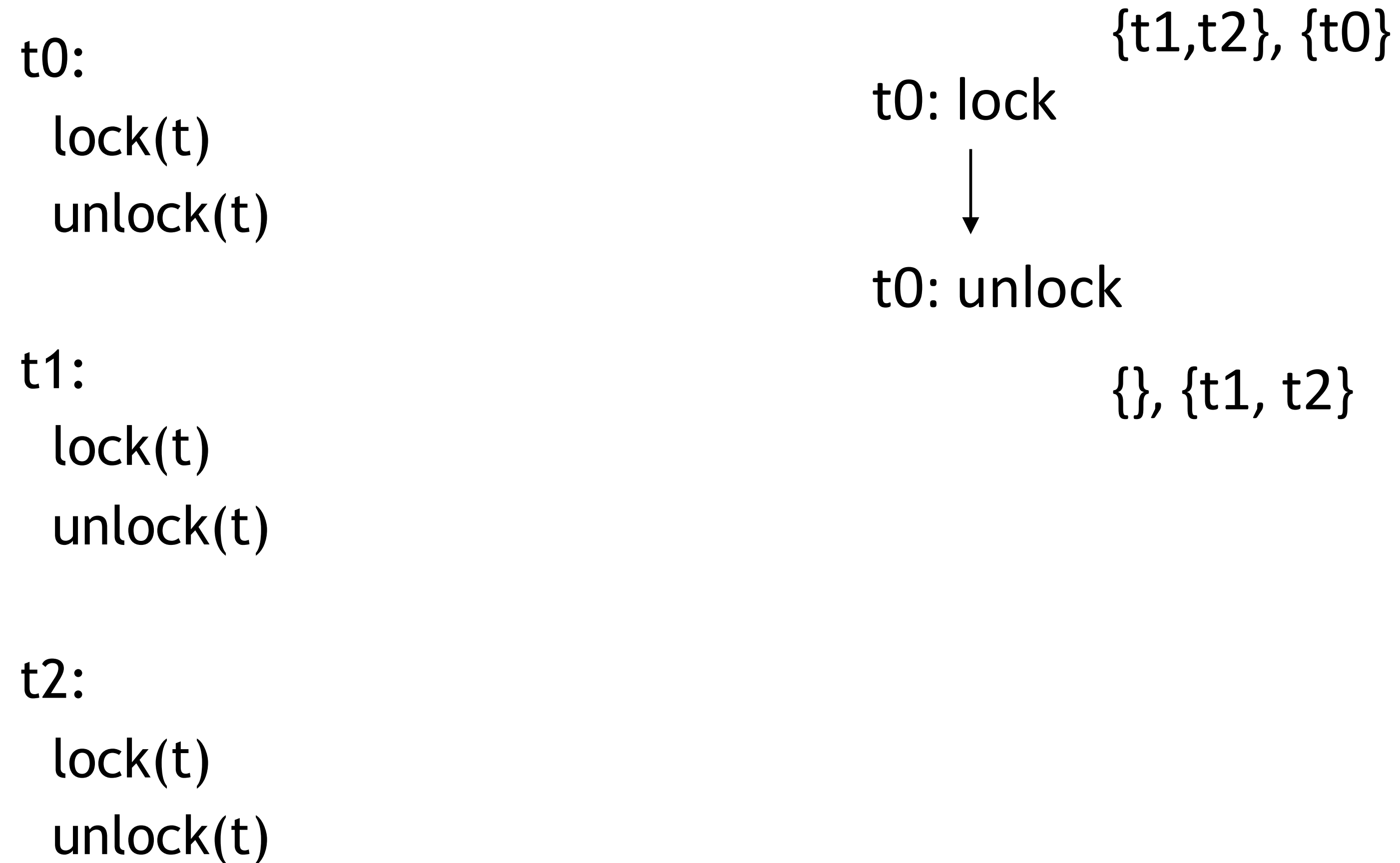
t2: unlock

↓

...

{ **BT** }, { **Done** }

A Simple DPOR Example



A Simple DPOR Example

{t2}, {t0,t1}

```
t0:
  lock(t)
  unlock(t)

t1:
  lock(t)
  unlock(t)

t2:
  lock(t)
  unlock(t)
```

{ **BT** }, { **Done** }

A Simple DPOR Example

t0:
  lock(t)
  unlock(t)

t1:
  lock(t)
  unlock(t)

t2:
  lock(t)
  unlock(t)

{t2}, {t0, t1}

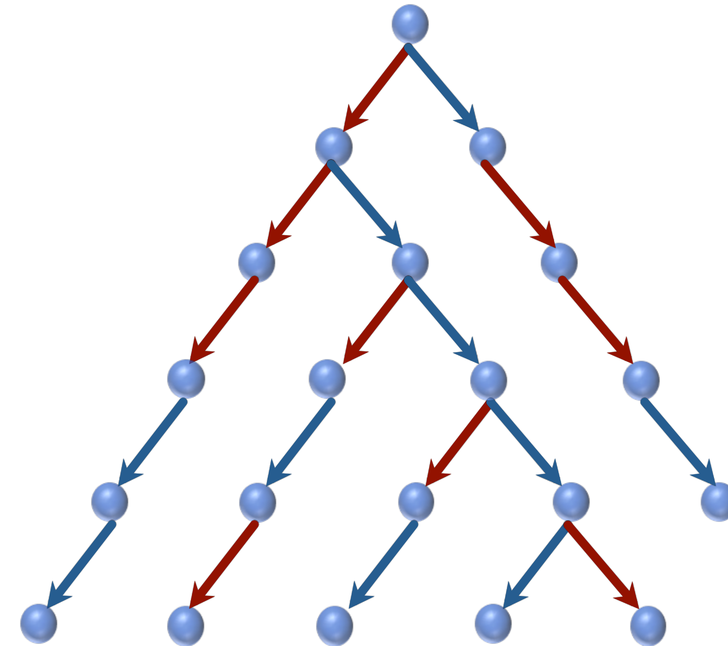
t1: lock

↓

t1: unlock

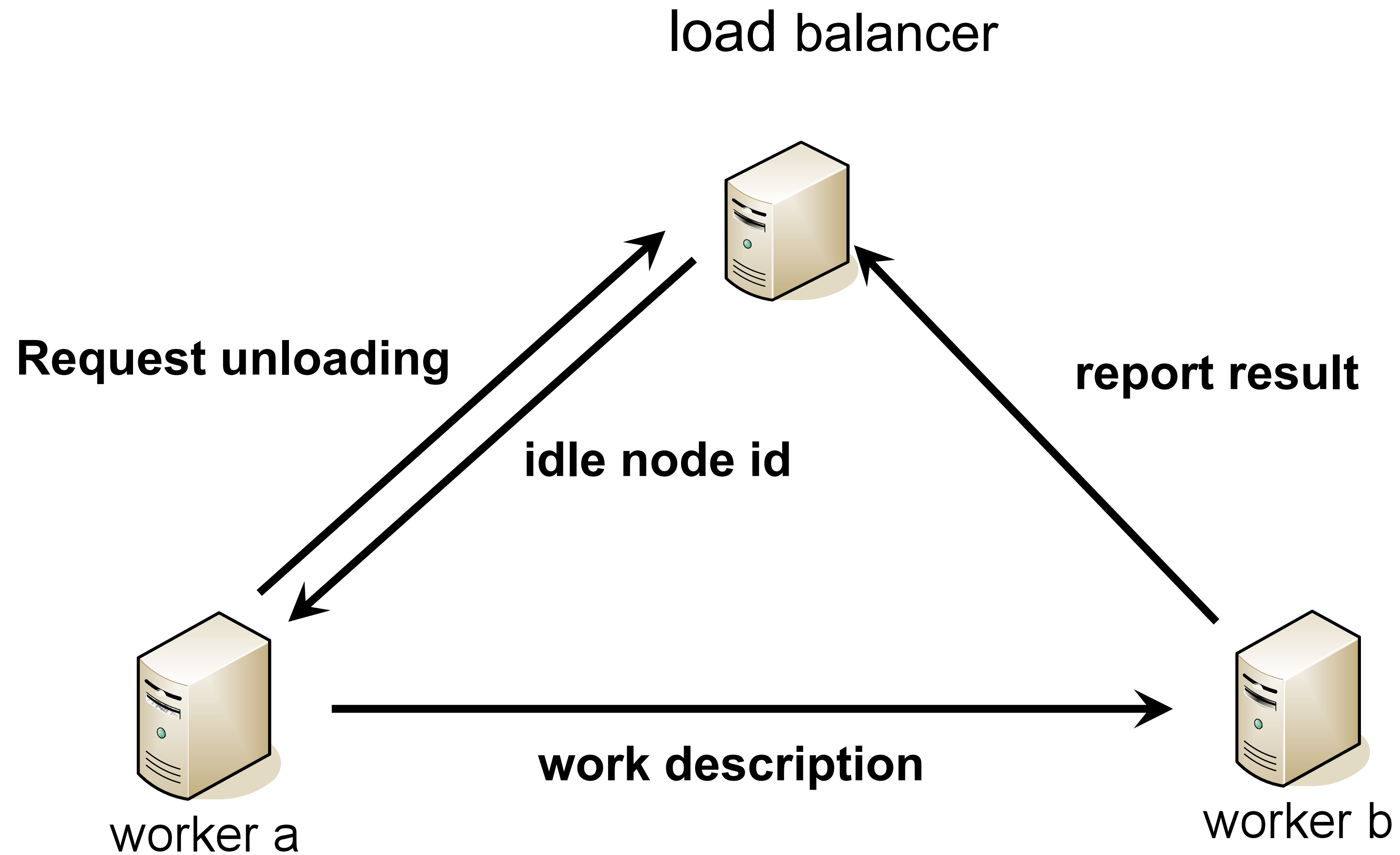
...

Observations



- Stateless model checking is ideal for "embarrassingly" parallelism
  - Different branches of an acyclic space can be explored concurrently
  - Simple master-slave scheme can work here (one load balancer + workers)

A Work Distribution Scheme

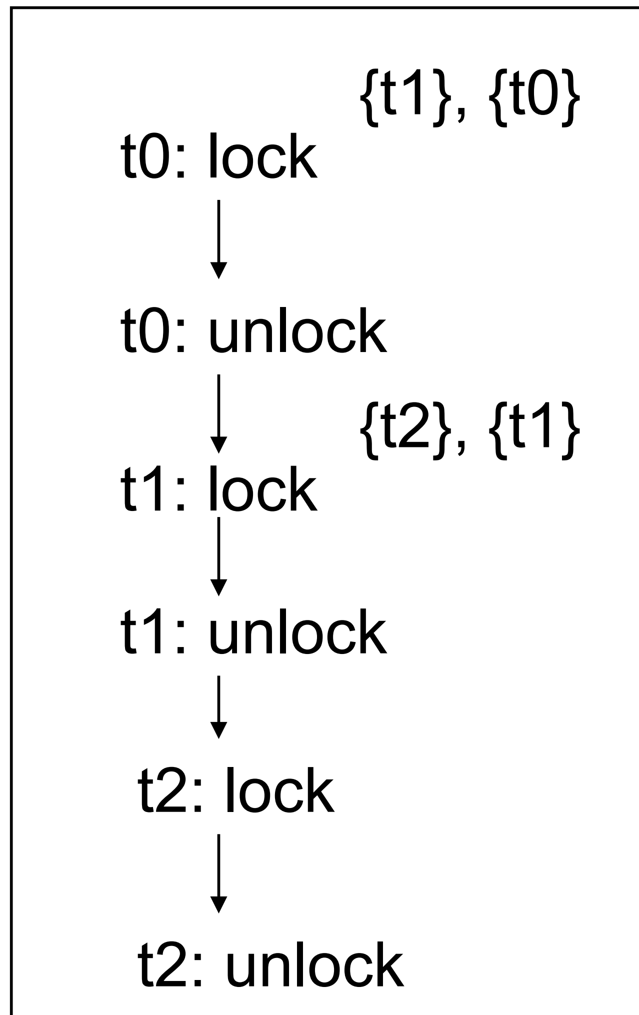


Initial implementation got little speedup

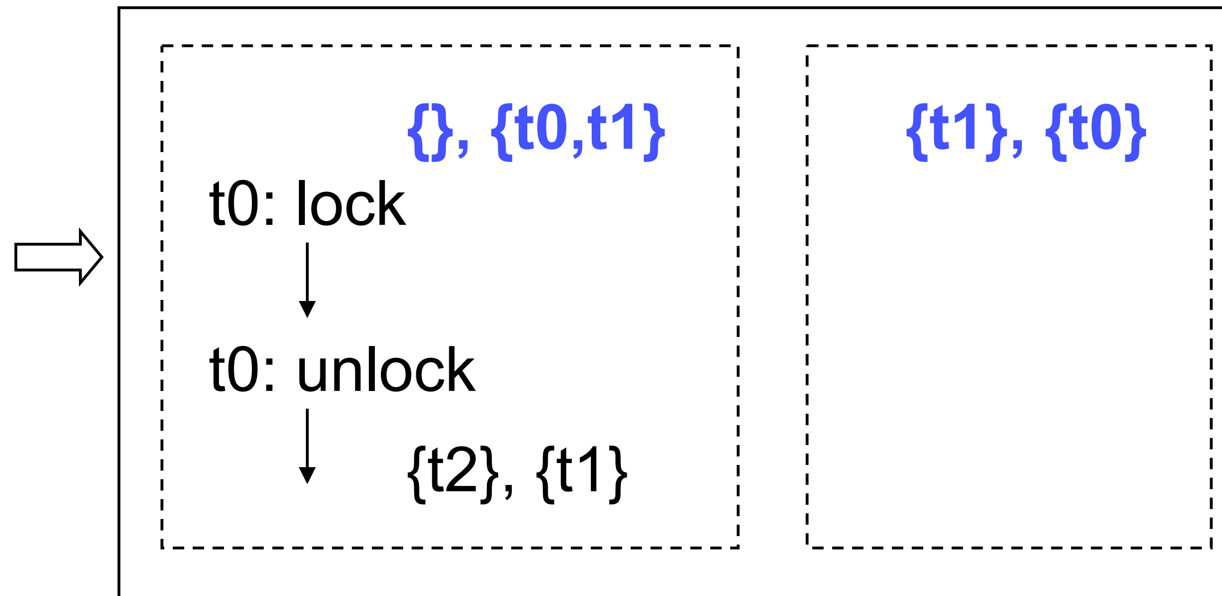
- Why?
  - DPOR algorithm does not fit into this parallel paradigm well
  - may have multiple nodes explore the same interleaving, result in redundant work

Illustration of the problem

One node:



Two nodes:



Heuristic : Handoff DEEPEST backtrack point for another node to explore

Reason : Largest number of paths emanate from there

Illustration of the problem

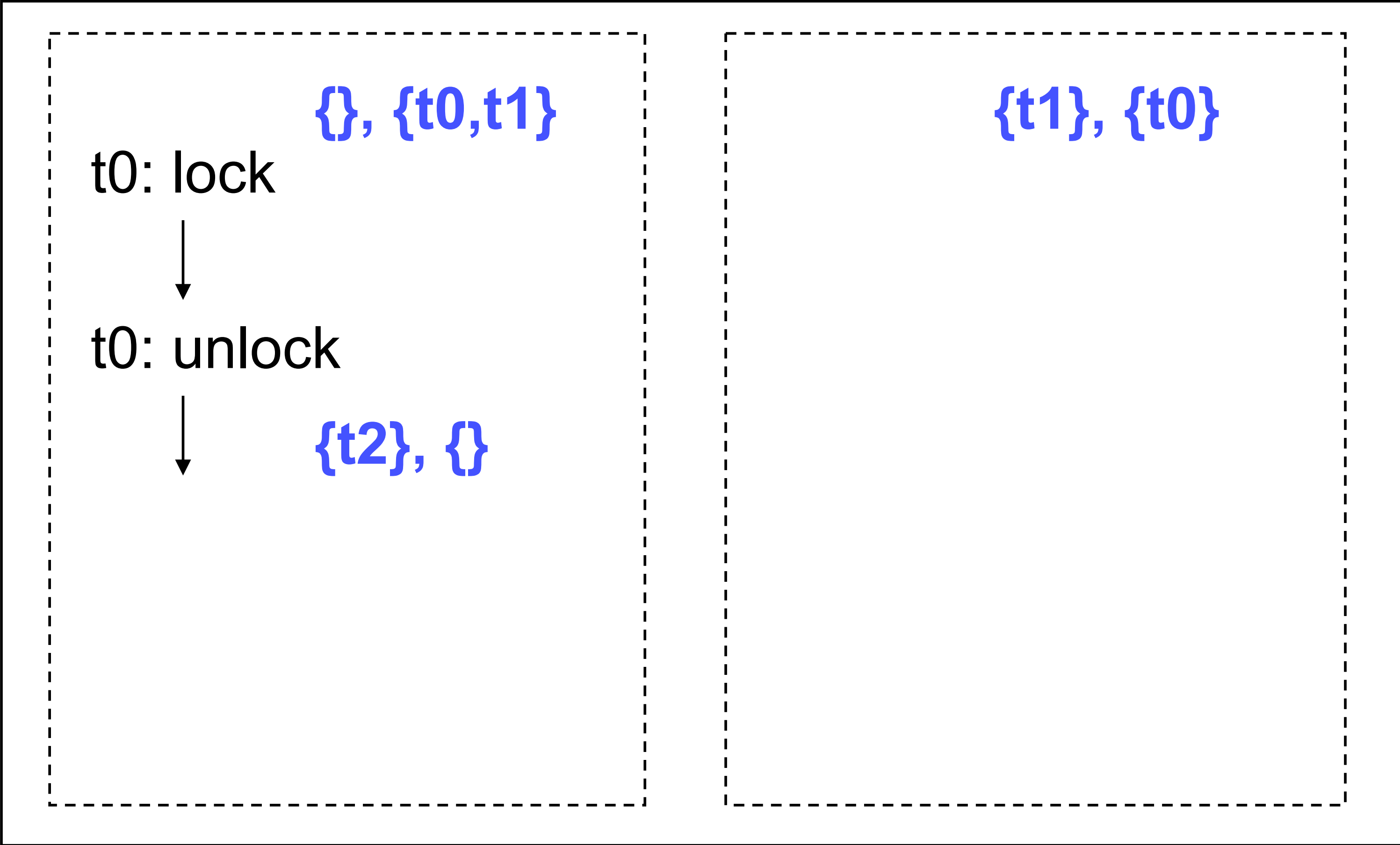


Illustration of the problem

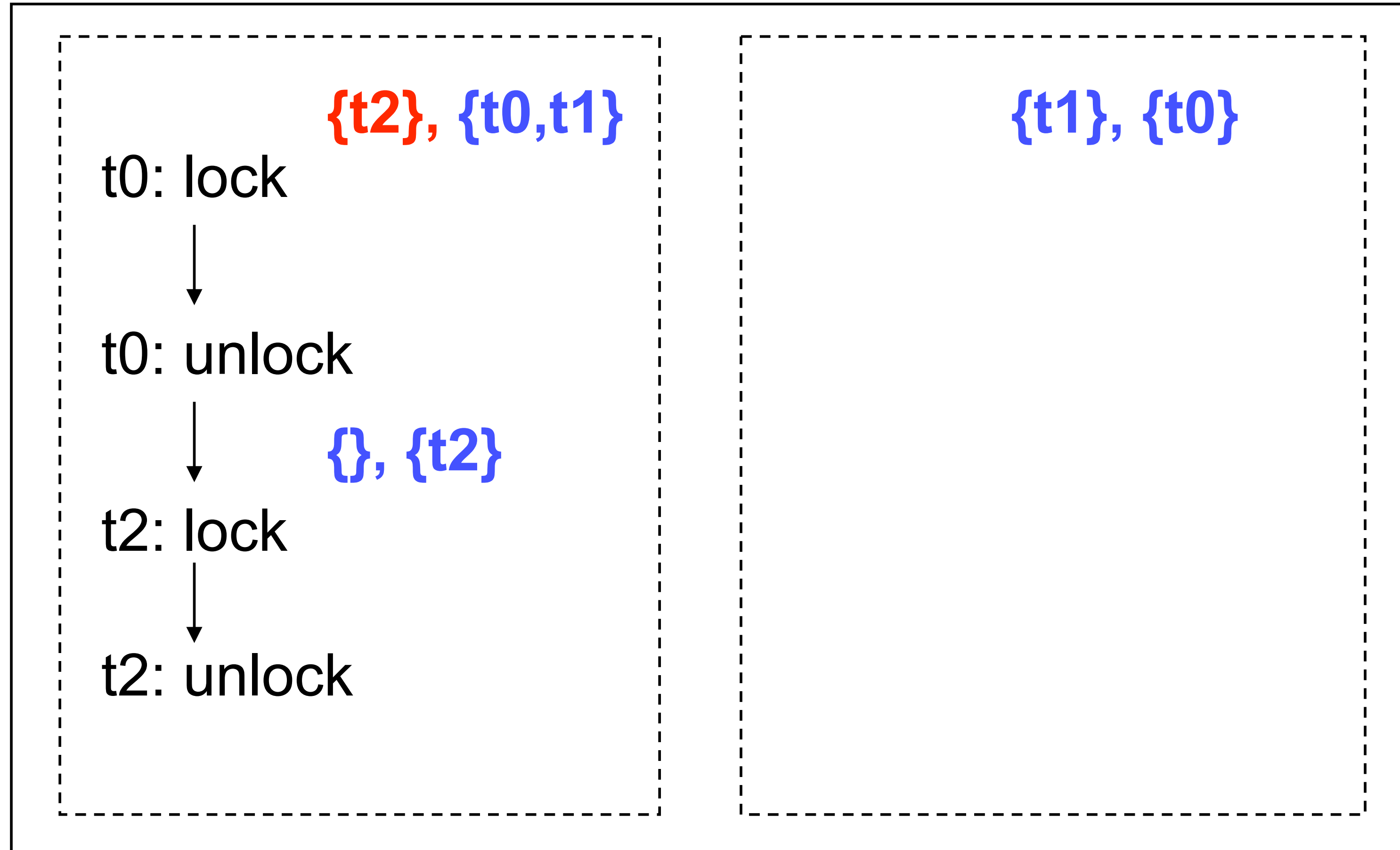


Illustration of the problem

Two nodes:

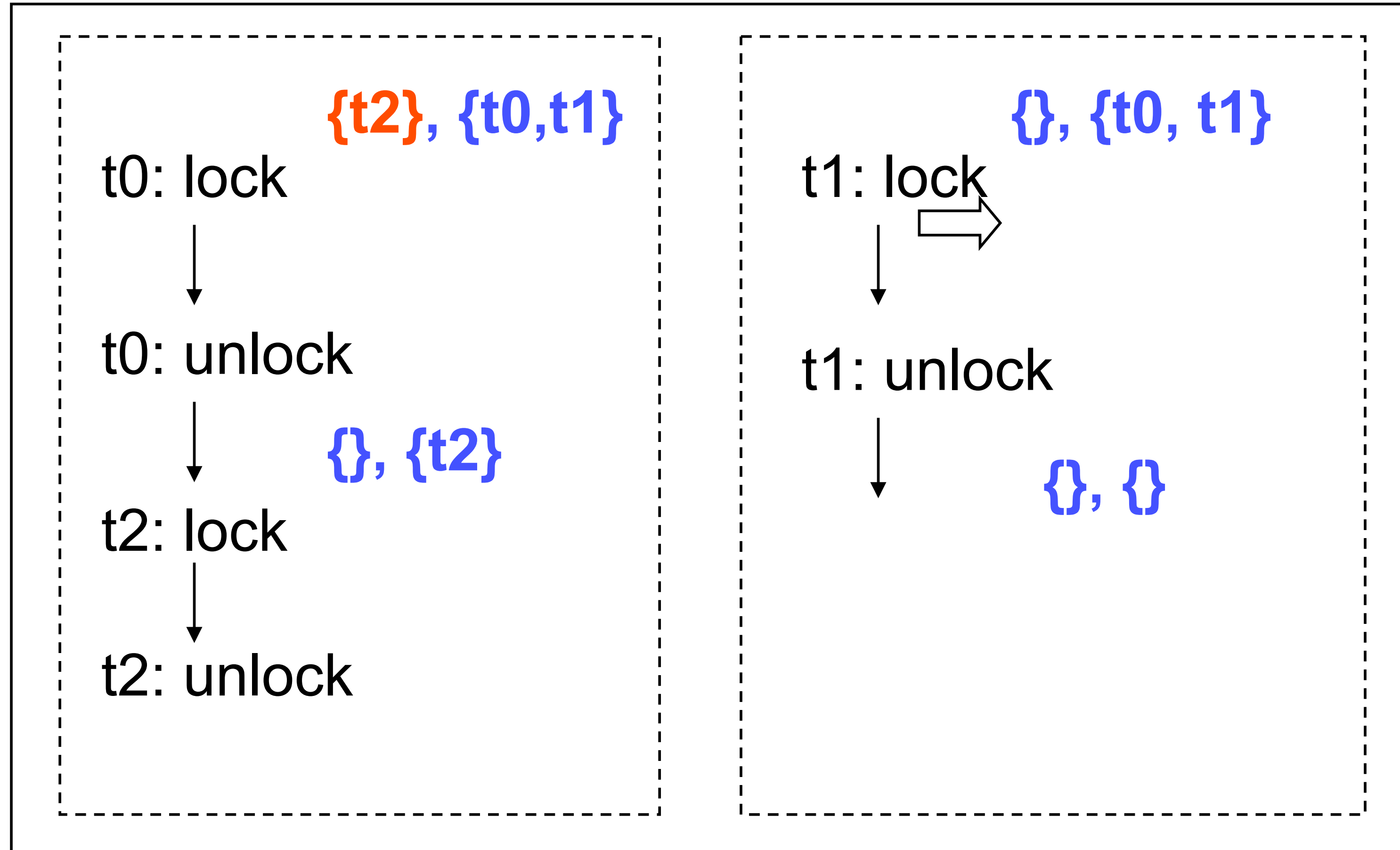


Illustration of the problem

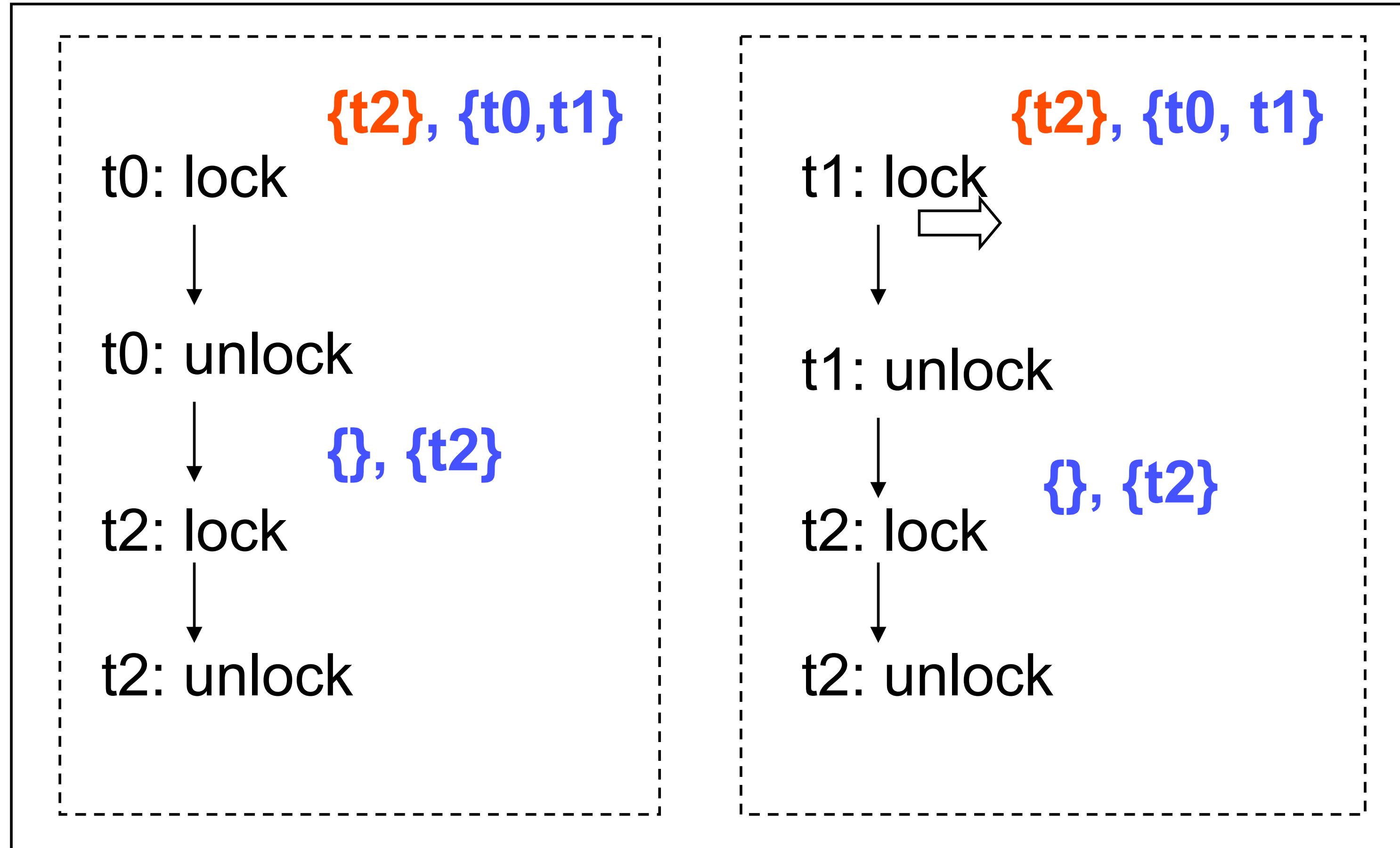


Illustration of the problem

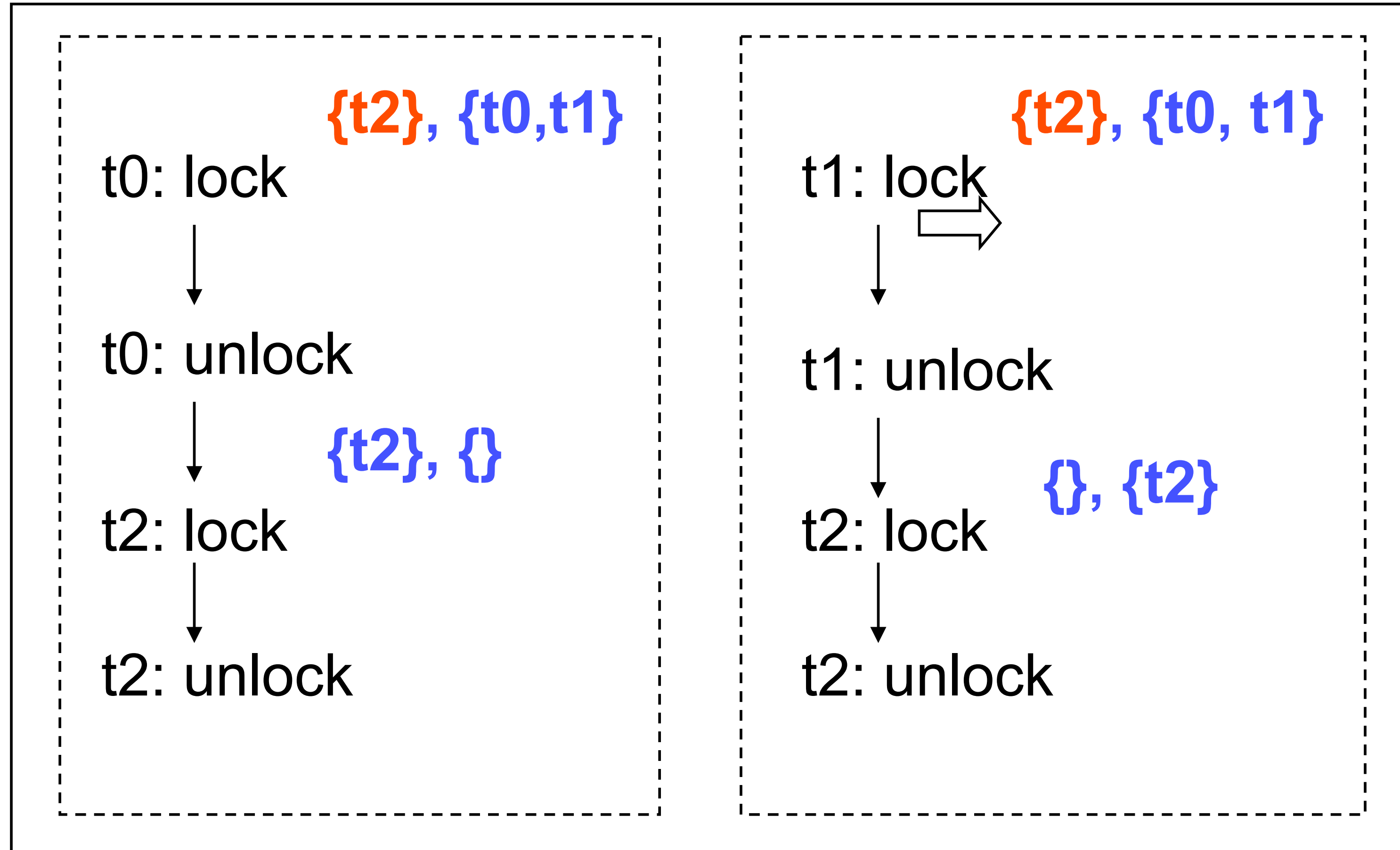
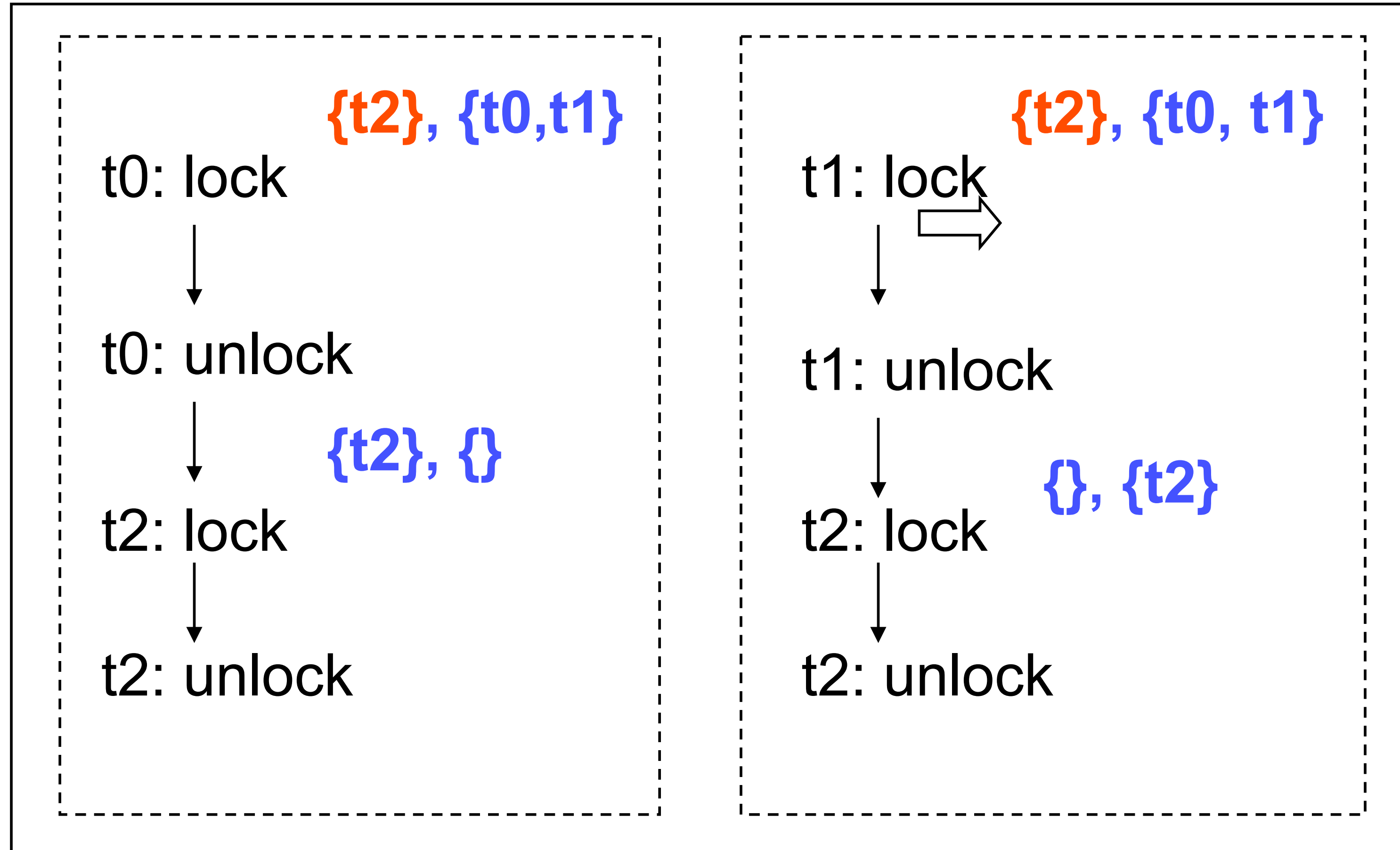
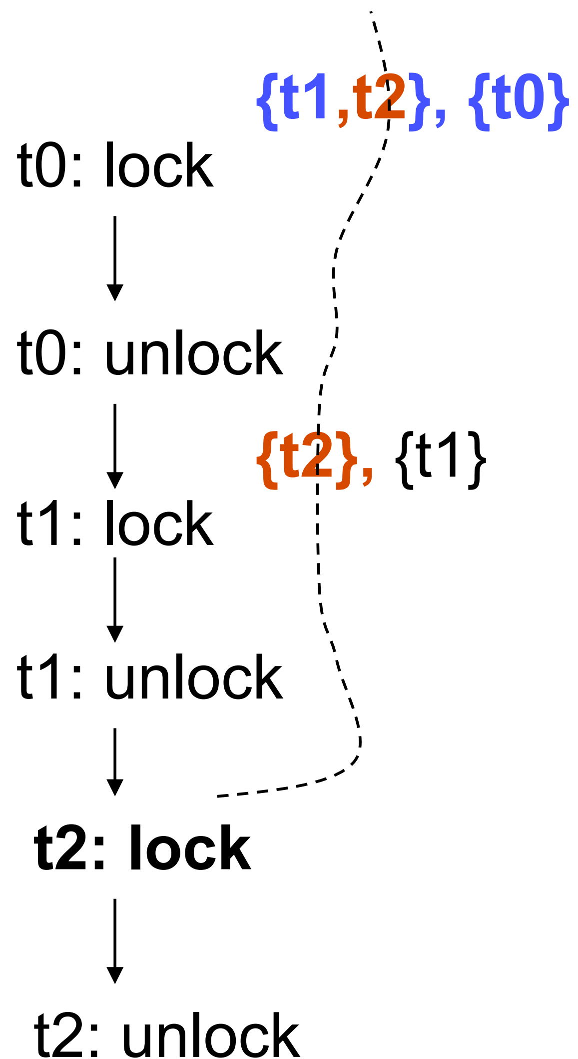


Illustration of the problem



New Backtrack Set Computation



Aggressively mark up the stack!

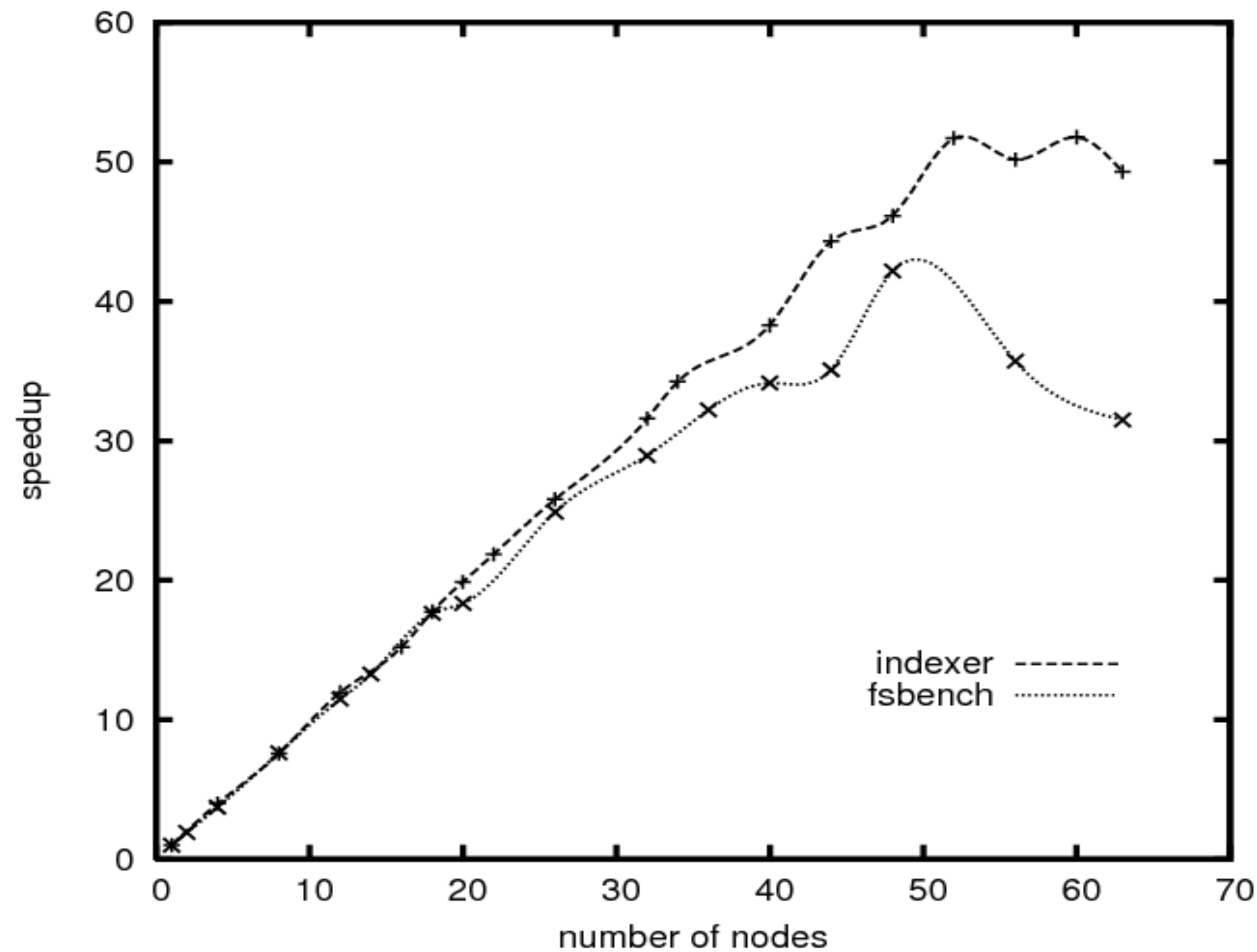
- Update the backtrack sets of ALL dependent operations!
- Forms a good allocation scheme
- Does not involve any synchronizations
- Redundant work may still be performed
- Likelihood is reduced because a node aggressively "owns" one operation and all its dependants

Implementation and Evaluation

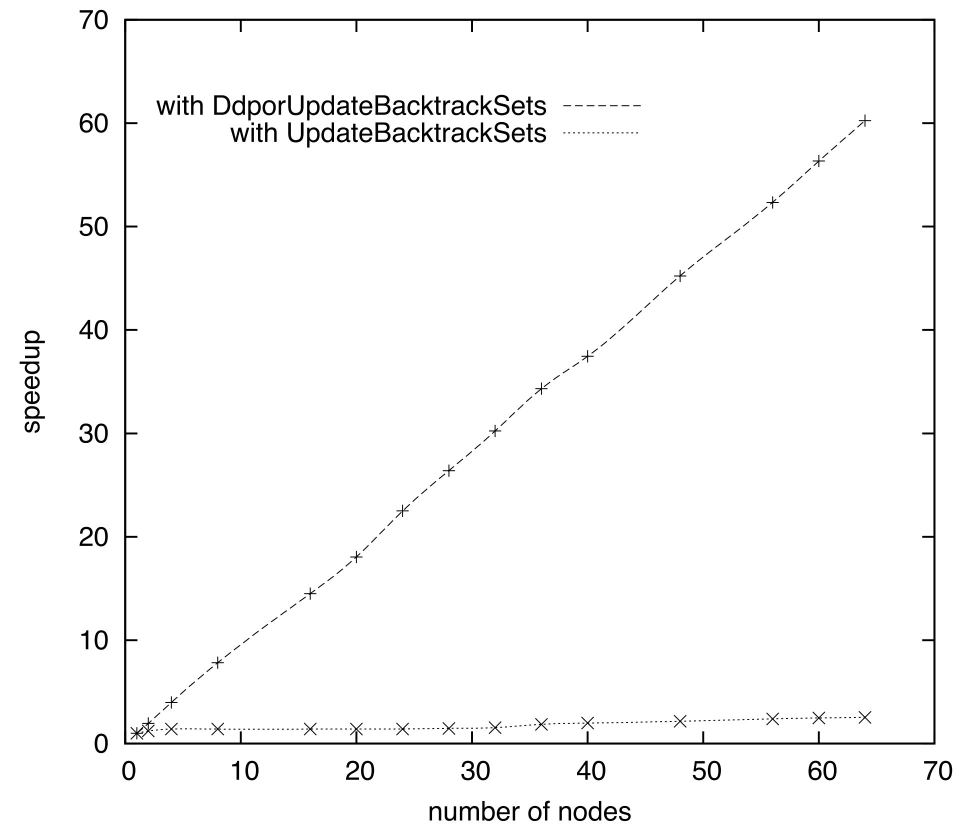
- Using MPI for communication among nodes
- Did experiments on a 72-node cluster
  - 2.4 GHz Intel XEON process, 2GB memory /node
  - Two (small) benchmarks

    Indexer & file system benchmark used in Flanagan and Godefoid's DPOR paper
  - aget -- a multithreaded ftp client
  - bbuf – an implementation of bounded buffer

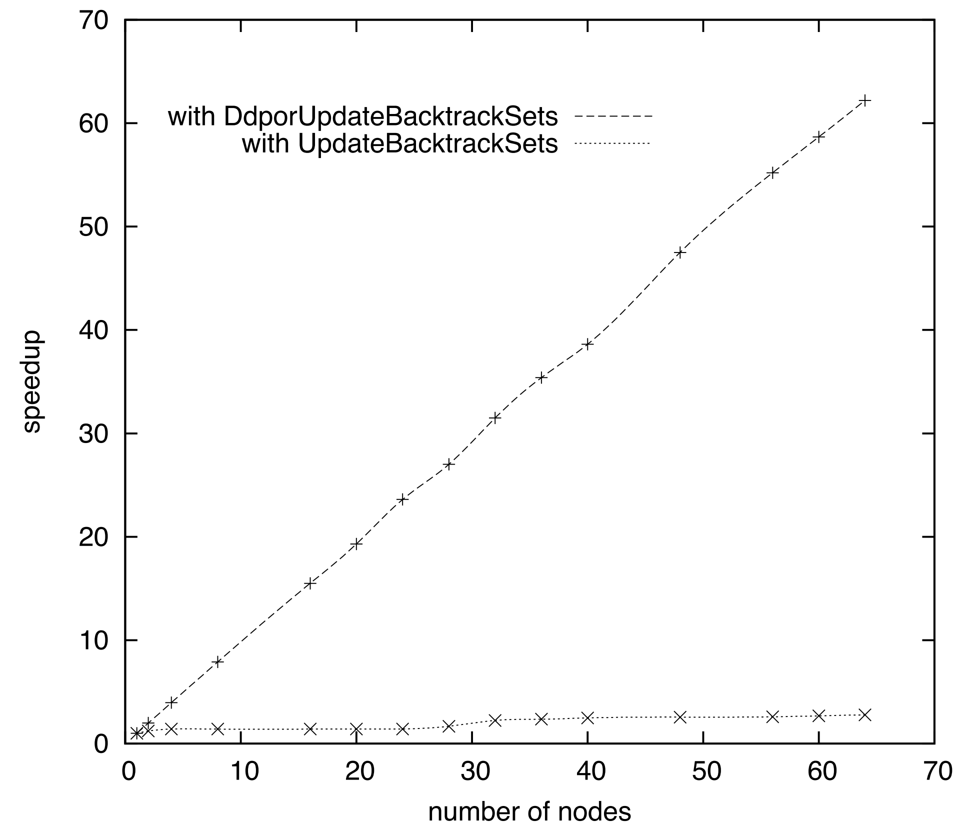
Speedup on indexer & fsbench (small examples); so diminishing returns > 40 nodes…



Speedup on aget



Speedup on bbuf



Combined Thread / Message Passing Verification

- Tool for Multicore Communications API (MCAPI) is under construction
- See http://www.multicore-association.org

- Extending ISP to handle MPI_THREAD_MULTIPLE planned

Pedagogical Material

- All Examples of MPI book of Pacheco being "solved" using ISP

http://www.cs.utah.edu/formal_verification/geof/pacheco/PachecoTests.html

Similarly we are assembling course material of our examination of all examples of the Herlihy / Shavit book using the MSR tool CHESS

For Inspect, we are assembling a case study of verifying a work-stealing queue

End of F